TESIS – KI 142502

PERBAIKAN MEKANISME LOAD BALANCING UNTUK KOMPUTASI KLASTER DENGAN REINFORCEMENT LEARNING PADA KONDISI DINAMIS

MOHAMMAD ZARKASI
5113201050

DOSEN PEMBIMBING
Waskitho Wibisono, S.Kom., M.Eng., Ph.D

PROGRAM MAGISTER
BIDANG KEAHLIAN KOMPUTASI BERBASIS JARINGAN
JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKHNOLOGI INFORMASI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA
2017

TESIS – KI142502

PERBAIKAN MEKANISME *LOAD BALANCING* UNTUK KOMPUTASI KLASTER DENGAN *REINFORCEMENT LEARNING* PADA KONDISI DINAMIS

Mohammad Zarkasi
NRP. 5113201050

DOSEN PEMBIMBING
Waskitho Wibisono, S.Kom., M.Eng., Ph.D

PROGRAM MAGISTER
BIDANG KEAHLIAN KOMPUTASI BERBASIS JARINGAN
JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA
2017

TESIS – KI142502

IMPROVEMENT OF LOAD BALANCING MECHANISM FOR CLUSTER COMPUTING USING REINFORCEMENT LEARNING ON DINAMIC CONDITION

Mohammad Zarkasi
NRP. 5113201050

SUPERVISOR
Waskitho Wibisono, S.Kom., M.Eng., Ph.D

MAGISTER PROGRAMME
INFORMATICS ENGINEERING DEPARTMENT
FACULTY OF INFORMATION TECHNOLOGY
SEPULUH NOPEMBER INSTITUTE OF TECHNOLOGY
SURABAYA
2017

Tesis disusun untuk memenuhi salah satu syarat memperoleh gelar
Magister Komputer (M.Kom.)
di
Institut Teknologi Sepuluh Nopember Surabaya

oleh:
Mohammad Zarkasi
Nrp. 5113201050

Dengan judul:
Perbaikan Mekanisme Load Balancing Untuk Komputasi Klaster Dengan Reinforcement Learning Pada Kondisi Dinamis

Tanggal Ujian : 12-01-2017
Periode Wisuda : 2016 Gasal

Disetujui oleh:

Waskitho Wibisono, S.Kom., M.Eng., Ph.D
NIP. 197410222000031001 (Pembimbing 1)

Dr.Eng. Radityo Anggoro, S.Kom., M.Sc
NIP. 1984101620081210002 (Penguji 1)

Tohari Ahmad, S.Kom., MIT, Ph.D
NIP. 197505252003121002 (Penguji 2)

Royyana Muslim I, S.Kom, M.Kom, Ph.D
NIP. 197708242006041001 (Penguji 3)

an. Direktur Program Pascasarjana
Asisten Direktur

Prof. Dr. Ir. Tri Widjaja, M.Eng.
NIP. 19611021 198603 1 001

Direktur Program Pascasarjana,

Prof. Ir. Djauhar Manfaat, M.Sc., Ph.D.
NIP. 19601202 198701 1 001

Perbaikan Mekanisme *Load Balancing* Untuk Komputasi Klaster Dengan *Reinforcement Learning* Pada Kondisi Dinamis

Nama Mahasiswa : Mohammad Zarkasi

NRP : 5113 201 050

Pembimbing : Waskitho Wibisono, S.Kom., M.Eng., Ph.D

ABSTRAK

Berbagai strategi penjadwalan dan *load balancing* adaptif telah banyak diterapkan di berbagai bidang. Strategi adaptif memberikan nilai lebih pada suatu sistem yang terpasang pada lingkungan dengan kondisi yang dinamis yang baru dijumpai pada saat suatu sistem tersebut diterapkan di lapangan. Salah satu bidang yang membutuhkan strategi *load balancing* yang baik adalah bidang komputasi klaster, seperti pada *Java Parallel Processing Framework* (JPPF). JPPF memiliki dua jenis strategi *load balancing*, yaitu strategi statis dan strategi dinamis. Strategi dinamis pada JPPF menggunakan metode distribusi normal. *Load balancing* dengan metode distribusi normal berjalan di *server* JPPF untuk menentukan jumlah *task* yang diberikan ke suatu *node*. *Load balancing* dinamis pada JPPF menggunakan parameter rata-rata waktu eksekusi sebuah *task*.

Dalam penelitian ini diusulkan penggunaan metode adaptif menggunakan algoritma *Reinforcement Leanring* (RL) sebagai metode *load balancing* pendistribusian *task*. RL banyak diterapkan di bidang robotika dan bidang-bidang lain yang membutuhkan kemampuan beradaptasi terhadap lingkungan. Dalam penelitian juga diusulkan beberapa parameter baru yang digunakan untuk menentukan jumlah *task* yang diberikan ke suatu *node*. Parameter baru tersebut adalah *throughput* jaringan antara *node* dan *server*, dan beban CPU di suatu *node*.

Hasil uji coba dari metode *load balancing* adaptif yang diusulkan di dalam penelitian ini adalah mampu memberikan *speedup* sebesar 45% dibandingkan dengan metode dinamis pada kondisi sumber daya jaringan dan CPU tidak mengalami beban. Sedangkan, pada kondisi sumber daya jaringan dan CPU

mengalami beban, metode *load balancing* adaptif yang diusulkan mampu memberikan *speedup* sebesar 21% dibandingkan dengan metode dinamis. Dengan mekanisme *load balancing* yang lebih baik, maka kinerja komputasi klaster secara keseluruhan akan meningkat.

Kata Kunci: *beban kerja, job, load balancing, metode adaptif, metode dinamis, penjadwalan, reinforcement learning, task, throughput*

Improvement of Load Balancing Mechanism for Cluster Computing Using Reinforcement Learning on Dinamic Condition

Student Name	: Mohammad Zarkasi
NRP	: 5113 201 050
Supervisor	: Waskitho Wibisono, S.Kom., M.Eng., Ph.D

ABSTRACT

Various adaptive scheduling and load balancing strategies has been widely applied in various fields. Adaptive strategy gives more value to a system when it is installed in a dynamic environment with new conditions encountered at the time a system is implemented in the field. One area that needs good scheduling strategies is cluster computing, such as the Java Parallel Processing Framework (JPPF). JPPF has two types of load balancing strategies, i.e. static strategy and dyanmic strategy. Dynamic strategy on JPPF uses normal distribution method. Load balancing that uses normal distribution methods run at JPPF server to determine the number of task assigned to a node. Dynamic load balancing at JPPF uses average execution time of a task as parameter.

In this research, an adaptive method using Reinforcement Learning (RL) is proposed as load balancing method on task distribution. RL is widely applied in robotic area and others that need adaptive capability against the environment. In this research also proposed some new parameters that are used to determine the number of tasks to send to the node. The new parameters are network throughput between node and server, and CPU load on node side.

The result of the proposed adaptive load balancing method in this research is able to provide speedup of 45% compared to dynamic method on conditions that network and CPU resources are not given any load. Meanwhile, on conditions that network and CPU resources are given some load, the proposed

adative load balancing method is able to provide a speedup of 21% compared to dynamic method. With better load balancing mechanism, the overall performance of cluster computing will increase.

Keywords**:** *adaptive method, dynamic method, scheduling, job, load balancing, reinforcement learning, scheduling, task, throughput, workload*

KATA PENGANTAR

Bismillahirohmanirohim

Alhamdulillahirobbil ‘alamin, segala puji bagi Allah Subhanahu wa ta’ala, yang telah memberikan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan Tesis yang berjudul “Perbaikan Mekanisme *Load Balancing* untuk Komputasi Klaster dengan *Reinforcement Learning* pada Kondisi Dinamis” dengan baik.

Dalam pelaksanaan dan pembuatan Tesis ini tentunya sangat banyak bantuan-bantuan yang penulis terima dari berbagai pihak, tanpa mengurangi rasa hormat penulis ingin mengucapkan terima kasih yang sebesar-besarnya kepada:

1. Allah SWT atas limpahan rahmat dan hidayah-Nya sehingga penulis dapat menyelesaikan Tesis ini dengan baik.
2. Kedua orang tua penulis, Bapak Imam Sujarot dan Ibu Musyarofah yang telah memberikan dukungan moral, spiritual dan material, semangat dan sabar kepada penulis serta selalu memberikan doa yang tiada habisnya yang dipanjatkan untuk penulis.
3. Bapak Waskitho Wibisono, S.Kom., M.Eng., Ph.D. selaku dosen pembimbing, yang telah memberikan kepercayaan, dukungan, bimbingan, nasehat, perhatian serta semua yang telah diberikan kepada penulis.
4. Teman-teman seperjuangan bidang minat NCC.
5. Teman-teman S2 Teknik Informatika ITS angkatan 2013.
6. Juga tidak lupa kepada semua pihak yang belum sempat disebutkan satu persatu yang telah membantu terselesaikannya tesis ini.

“Tiada gading yang tak retak,” begitu pula dengan Tesis ini. Oleh karena itu penulis mengharapkan saran dan kritik yang membangun dari pembaca

Surabaya, Januari 2017

Penulis

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

[Halaman ini sengaja dikosongkan]

BAB I
PENDAHULUAN

1.1. Latar Belakang

Adanya teknologi jaringan komunikasi memungkinkan dua entitas untuk saling terhubung. Hal ini memungkinkan komputer-komputer untuk saling terhubung melalui jaringan komunikasi. Dengan semakin tingginya kecepatan pengiriman yang dapat dilakukan oleh teknologi komunikasi saat ini, hal ini telah memungkinkan komputer-komputer untuk saling berbagi sumber daya, seperti CPU, *memory* dan media penyimpanan, untuk menyediakan aplikasi yang lebih unggul dibandingkan dengan sistem tunggal (Berman, Wolski, Figueira, & Schopf, 1996). Hal ini juga didorong oleh permasalahan yang ingin diselesaikan telah menjadi lebih kompleks dan dengan skala yang lebih besar untuk dikerjakan oleh sebuah komputer tunggal (Mohammadpour, Sharifi, & Paikan, 2008).

Salah satu aplikasi yang memanfaatkan keuntungan dari kemajuan teknologi jaringan komunikasi ini adalah komputasi grid dan komputasi klaster. Komputasi grid adalah bentuk komputasi yang melibatkan banyak komputer yang biasanya bersifat heterogen dan tersebar di berbagai lokasi yang berbeda secara geografi. Sedangkan, komputasi klaster adalah komputasi yang melibatkan banyak komputer yang berada di satu tempat. Komputasi klaster merupakan salah satu komponen penyusun dari komputasi grid (Singh, 2005).

Di dalam komputasi klater dikenal istilah *server*, *node*, *client*, *job* dan *task*. *Task* merupakan unit komputasional yang biasanya merupakan sebuah program yang tidak dapat dipecah-pecah menjadi proses yang kecil. *Job* merupakan aktivitas komputasional yang terdiri atas satu atau beberapa *task* (Xhafa & Abraham, 2010). *Client* merupakan entitas yang menciptakan *job*, *server* merupakan entitas yang mendistribusikan *job* dan *node* adalah entitas yang melakukan tugas komputasi (Cohen, 2014).

Pada saat menerima *job*, terdapat kemungkinan komputer-komputer yang terdapat di dalam sistem komputasi paralel mengalami beban yang tidak

seimbang. Kondisi beban sistem yang tidak seimbang ini dapat menurunkan *Quality of Service* (QoS), sehingga diperlukan metode *load balancing*.

Di dalam komputasi klaster terdapat istilah *policy load balancing. Policy load balancing* di dalam komputasi klaster dapat dikategorikan menjadi empat, yaitu metode statis, dinamis, *hybrid* dan adaptif (Mahato, Maurya, Tripathi, & Singh, 2016). *Policy* statis mempertimbangkan status sistem dan aplikasi secara statis dan menerapkan informasi ini dalam pengambilan keputusan. Keuntungan utama dari *policy* statis adalah *overhead* yang lebih rendah karena pengambilan keputusan telah dikerjakan sebelum *job* diberikan (Patil & Gopal, 2012). *Policy* dinamis berjalan dengan memindahkan *job* dari komputer yang memiliki kelebihan beban ke komputer yang memiliki beban sedikit. Metode *hybrid* adalah metode *load balancing* yang menggabungkan metode statis dan metode dinamis. Sedangkan, metode adaptif adalah metode yang secara adaptif menyesuaikan porsi *task* menggunakan informasi terkini dari sistem dan suatu nilai *threshold* tertentu.

Policy adaptif untuk permasalahan *load balancing* telah banyak dilakukan penelitian. Mohammadpour mengusulkan *policy* adaptif dengan menggunakan informasi beban CPU, penggunaan memory dan lalu lintas jaringan untuk menentukan beban kerja tiap *node* dan mengkombinasikannya dengan *property* tiap *job*. Guo mengusulkan penggunaan nilai prediksi beban *node* untuk mempartisi sistem menjadi beberapa bagian secara adaptif yang mampu memberikan keseimbangan antara *througput* dan *jitter*. Sedangkan, pada *Java Parallel Processing Framework* (JPPF), algoritma *Reinforcement Learning* (RL) digunakan sebagai metode adaptif untuk mendistribusikan *job* ke *node-node* yang tergabung ke salam satu klaster. Pada penelitian yang lain, Mirza mengusulkan prediktor *throughput* pada jaringan *wireless* dengan menggunakan analisis *time series* dan teknik *machine learning*. Keuntungan menggunakan *policy* adaptif adalah kemampuan untuk menyesuaikan kondisi di lapangan yang mungkin berbeda dengan kondisi pada saat penelitian.

JPPF merupakan salah satu *framework* untuk pemrosesan komputasi klaster. JPPF dibangun di atas bahasa pemrograman Java dan dapat dijalankan di atas komputer dengan sistem operasi yang mendukung Java. Pemrograman berbasis

Java menjadi terkenal karena properti "*write once, run anywhere*" yang memungkinkan aplikasi yang dibangun dengan bahasa pemrograman Java tidak terikat pada *platform* tertentu dan memiliki kinerja yang kompetitif jika dibandingkan dengan bahasa pemrograman *High Performance Computing* (HPC).

Sebagai *framework* untuk komputasi paralel, JPPF dilengkapi dengan *scheduler* atau dapat juga disebut *load-balancer*. *Scheduler* bertugas untuk mengatur pendistribusian *task-task* ke *node-node* yang terhubung ke server JPPF. *Scheduler* yang terdapat di dalam JPPF memiliki beberapa strategi pendistribusian, di antaranya adalah distribusi secara manual dan adaptif. Distribusi secara manual membagi jumlah *task* untuk tiap *node* dalam jumlah yang tetap. Sedangkan distribusi secara adaptif membagi *task* untuk tiap *node* berdasarkan *feedback* kinerja tiap *node* terhadap *task-task* yang telah dikerjakan sebelumnya (Cohen, 2014).

Strategi distribusi adaptif pada JPPF menggunakan algoritma *reinforcement learning* (RL). RL berjalan dengan pengambilan *action* terhadap *environment* sehingga memaksimalkan *reward* (Lee, 2005). Pada distribusi adaptif, JPPF memberikan dua *feedback* berupa jumlah *task* yang telah dikerjakan oleh *node* dan total waktu yang digunakan untuk mengerjakan *task-task* tersebut (Cohen, 2014)

Dari kedua *feedback* yang diberikan oleh JPPF tersebut, RL menentukan jumlah *task* yang akan diberikan ke suatu *node*. Namun, beberapa permasalahan sering muncul pada jaringan komputer klaster karena jaringan komputer mengalami kondisi yang dinamis. Kondisi jaringan yang dinamis tidak termasuk dalam perhitungan algoritma pendistribusian. Beberapa kondisi dinamis pada jaringan komputer klaster antara lain *throughput* dari *server* ke suatu *node* mengalami penurunan kinerja dan beban prosesor pada suatu *node* sedang tinggi. Maka kondisi-kondisi tersebut tidak menjadi parameter perhitungan pendistribusian task pada JPPF.

Oleh karena itu, penulis mengajukan sebuah penelitian mekanisme *load-balancing* pada JPPF menggunakan algoritma RL dengan tambahan parameter berupa troughput jaringan dan beban prosesor. Hasil yang diharapkan dari penelitian ini adalah perbaikan mekanisme *load balancing* pada komputasi klaster dengan distribusi adaptif menggunakan algoritma RL pada kondisi yang dinamis.

1.1. Perumusan Masalah

Rumusan masalah pada penelitian ini adalah:

1. Bagaimana menambahkan metode *load balancing* baru ke dalam *framework* JPPF?
2. Bagaimana meningkatkan kinerja mekanisme *load balancing* pada JPPF menggunakan algoritma adaptif?

1.2. Hipotesis

Hipotesis yang diajukan dalam penelitian ini adalah "jika pendistribusian *task* pada komputasi klaster dengan strategi adaptif dengan mempertimbangkan *throughput* antara *server* dan *node*, beban prosesor dan ukuran *bundle*, maka akan meningkatkan kinerja *load balancing* pada kondisi yang dinamis."

1.3. Batasan Masalah

Batasan masalah pada penelitian ini adalah sebagai berikut:

1. Penelitian dilakukan dengan menggunakan JPPF.
2. Kode program ditulis dalam Bahasa Java.
3. Tiga parameter baru yang diteliti adalah *throughput* antara *server* dan *node*, beban kerja prosesor pada *node* dan ukuran *bundle*.
4. Entitas yang digunakan pada penelitian terdiri atas sebuah *client* JPPF, sebuah *server* JPPF dan empat buah *node* JPPF.
5. Kasus uji coba yang digunakan adalah proses dekripsi *ciphered* MD5.

1.4. Tujuan

Berdasarkan perumusan masalah yang dijelaskan pada subbab 1.1, penelitian ini bertujuan untuk:

1. Mengetahui cara menambahkan *load balancing* baru ke dalam JPPF.
2. Meningkatkan kinerja *load balancing* pada JPPF menggunakan distribusi adaptif.

1.5. Manfaat

Adapun manfaat dari penelitian ini adalah untuk meningkatkan kinerja *load-balancing* pada kondisi *throughput* jaringan dan CPU mengalami beban yang

berubah-ubah, sehingga dapat ditentukan jumlah *task* yang optimal untuk dikirimkan ke suatu *node*.

1.6. Kontribusi Penelitian

Kontribusi dari penelitian ini adalah peningkatan kinerja mekanisme *load balancing* menggunakan strategi distribusi adaptif berbasis algoritma RL pada kondisi yang dinamis, antara lain: *throughput* yang berubah-ubah antara *node* dengan *server*; dan beban prosesor pada suatu *node*.

1.7. Sistematika Penulisan

Sistematika penulisan pada penelitian ini adalah:

1. BAB I PENDAHULUAN. Bab ini berisi pendahuluan yang menjelaskan latar belakang, perumusan masalah, tujuan, hipotesis, batasan masalah, tujuan, manfaat, kontribusi penelitian dan sistematika penulisan terkait dengan penelitian.
2. BAB II DASAR TEORI DAN KAJIAN PUSTAKA. Bab ini berisi tinjauan pustaka yang mengacu pada dasar-dasar teori terkait penelitian.
3. BAB III METODOLOGI PENELITIAN. Bab ini berisi perancangan dan implementasi penelitian serta pembahasan analisis mengenai hasil implementasi penelitian.
4. BAB IV HASIL DAN PEMBAHASAN. Bab ini berisi tentang tahapan pengujian dan pembahasan hasil yang dilakukan. Tahapan pengujian meliputi skenario pengujian, variabel pengujian dan lingkungan pengujian. Pembahasan hasil meliputi analisis hasil yang didapat dari pengujian secara nyata.
5. BAB V HASIL DAN PEMBAHASAN. Bab ini berisi kesimpulan dan saran untuk pengembangan tesis.

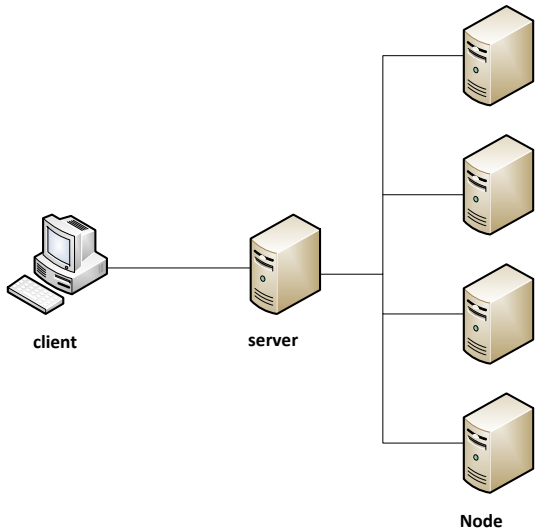
[Halaman ini sengaja dikosongkan]

BAB II
DASAR TEORI DAN KAJIAN PUSTAKA

Pada Bab II ini dijelaskan dasar teori dan kajian pustaka yang digunakan sebagai landasan ilimiah penelitian. Kajian pustaka yang dilakukan mencakup komputasi klaster dan metode pendistribusian untuk komputasi klaster.

2.1. Komputasi Klaster

Pada awalnya, perangkat lunak ditulis untuk komputasi serial. Komputasi serial berjalan dengan cara memecah permasalahan menjadi instruksi-instruksi yang berurutan. Instruksi-instruksi tersebut dieksekusi secara berurutan satu persatu. Biasanya eksekusi dilakukan pada prosesor tunggal dan hanya satu instruksi yang dieksekusi pada satu waktu (Barney, 2016).



Gambar 2.1 Arsitektur komputasi klaster

Pendekatan pertama dalam komputasi paralel adalah dengan membuat suatu komputer yang terdiri atas banyak prosesor. Namun, pendekatan ini membutuhkan biaya yang cukup besar. Pendekatan yang lain adalah dengan menggunakan beberapa komputer yang dihubungkan oleh jaringan LAN. Komputasi paralel berada di area komputasi pada mesin tunggal dengan banyak prosesor atau prosesor dengan inti banyak. Sedangkan komputasi dengan melibatkan banyak komputer untuk saling bekerja menangani task yang sama disebut komputasi

klaster (Barney, 2016). Gambar 2.1 menunjukkan arsitektur komputasi klaster secara umum.

Komputasi klaster terbentuk beberapa komputer yang memiliki spesifikasi yang sama atau hampir sama, yang terletak pada suatu tempat yang dihubungkan oleh koneksi jaringan LAN yang cepat. Komputasi klaster bertujuan untuk menyediakan *availability*, *reliability*, dan *scalability* yang lebih jika dibandingkan dengan sistem tunggal (Microsoft, 2013). Pada komputasi grid, setiap *node* mungkin mengeksekusi *task* yang berbeda. Namun, pada komputasi klaster, tiap *node* mengerjakan *task* yang sama.

Beberapa keuntungan mengunakan komputasi klaster adalah dapat menghemat uang dan waktu. Secara teori, mengalihkan komputasi yang membutuhkan waktu komputasi lama ke beberapa komputer akan mempersingkat waktu yang diperlukan untuk menyelesaikan suatu permasalahan. Sebagai contoh, pada suatu program terdapat suatu *procedure p* yang menggunakan 90% waktu eksekusi program tersebut. Tingginya waktu eksekusi pada *procedure p* tersebut dapat disembunyikan dengan memecah *procedure* tersebut menjadi 10 *subprocedure* yang masing-masing akan memakan waktu eksekusi sebesar 9%. Dengan waktu yang lebih singkat, maka biaya yang digunakan juga semakin kecil. Selain itu, komputasi klaster juga dapat menyelesaikan permasalahan yang rumit yang tidak dapat diselesaikan oleh komputasi dengan komputer tunggal, terutama pada komputer dengan RAM yang terbatas (Barney, 2016).

2.2. Penjadwalan Pada Komputasi Klaster

Permasalahan penjadwalan pada komputasi klaster dan komputasi *grid* telah menjadi topik penelitian oleh banyak peneliti. Penjadwalan *job* sendiri menjadi komponen yang paling penting dalam komputasi klaster. Pada lingkungan komputasi *grid*, penjadwalan dikatakan memiliki kerumitan komputasi *NP-hard* (Xhafa & Abraham, 2010).

Dengan kerumitan komputasi *NP-hard*, Xhafa mengajukan metode *heuristic* dan *meta-heuristic* untuk melakukan penjadwalan *job* pada komputasi *grid* (Xhafa & Abraham, 2010). Metode *heuristic* dipilih karena tidak memerlukan solusi yang

optimal, namun memungkinkan mendapatkan solusi yang efisien dalam waktu yang singkat.

Dengan permasalahan penjadwalan sebagai permasalahan yang rumit, para peneliti bergerak ke bidang *NP-completeness*, algoritma *approximation*, dan metode *heuristic* untuk memperoleh solusi yang optimal. *First-come-first-serve* (FCFS) merupakan penjadwalan *job* yang sederhana yang sering ditemukan pada sistem komputasi klaster yang nyata di lapangan, namun para peneliti biasanya menilai metode tersebut kurang memadai (Schwiegelshohn & Yahyapour, 1998). Pada JPPF, FCFS direpresentasikan oleh strategi distribusi manual. Karena kinerja penjadwalan *job* pada komputasi klaster sangat bergantung kepada *workload* (beban kerja), penjadwalan adaptif memungkinkan hasil terbaik untuk permasalahan penjadwalan.

Menurut Feitelson (Feitelson & Rudolph, 2006), matriks yang paling penting untuk *interactive job* adalah *response time*. Sedangakan matriks *system utilization* adalah matriks yang paling penting untuk *batch job*.

2.1. Throughput jaringan

Parameter *throughput* jaringan menunjukkan kecepatan transfer data yang dapat dilakukan antara *server* dengan *node*. Hal ini menjadi penting karena ada suatu kondisi dalam jaringan berupa *troughput* yang berbeda pada tiap-tiap komputer. Perbedaan itu dapat disebabkan oleh perbedaan perangkat keras yang digunakan ataupun dari sisi aplikasi. Parameter *troughput* ini diperoleh dengan menghitung jumlah waktu yang digunakan untuk mengirimkan job data, baik pada saat mengirimkan maupun menerima hasil job. Untuk menghitung throughput digunakan persamaan berikut:

$$Throughput\ (\frac{b}{s}) = \frac{amount\ of\ data\ (bits)}{transmission\ time\ (\mu s)} \qquad \textbf{(2.1)}$$

Salah satu metode untuk memprediksi *throughput* pada transfer TCP dalam jumlah besar adalah dengan menggunakan pengukuruan *throughput* pada transfer sebelumnya dalam *path* yang sama. Metode prediksi *History-Based* (HB) hampir sama dengan *time series forecasting* tradisional, yaitu sampel-sampel sebelumnya

dari proses acak yang tidak diketahui digunakan untuk memprediksi nilai dari proses yang akan datang. Metode HB memungkinkan untuk diterapkan pada kasus transfer TCP dalam jumlah besar yang terjadi berulang-ulang pada path yang sama (He, Dovrolis, & Ammar, 2005).

Moving Average (MA) merupakan salah satu dari keluarga *simple linear predictor*. MA dirumuskan dengan

$$X_{i+1} = \frac{1}{n} \sum_{k=i-n+1}^{i} X_k \qquad \textbf{(2.2)}$$

dengan X_{i+1} adalah nilai yang diprediksi, $X_i$adalah nilai aktual yang diamati. Jika n terlalu kecil, predictor tidak dapat menghasilkan prediksi yang baik, sedangkan apabila $n$terlalu besar, predictor tidak dapat beradaptasi dengan tepat terhadap *non-stationarities*.

2.2. Beban kerja prosesor

Beban kerja suatu *node* menunjukkan beban komputasi yang sedang ditangani oleh suatu node. Beban kerja tersebut dapat disebabkan oleh adanya aplikasi-aplikasi lain yang juga berjalan di suatu node. Beban kerja juga dapat disebabkan oleh adanya task lain yang sedang dikerjakan oleh node. Apabila beban kerja suatu node sedang tinggi, maka akan berpengaruh terhadap kinerja komputasi node tersebut terhadap task yang diberikan.

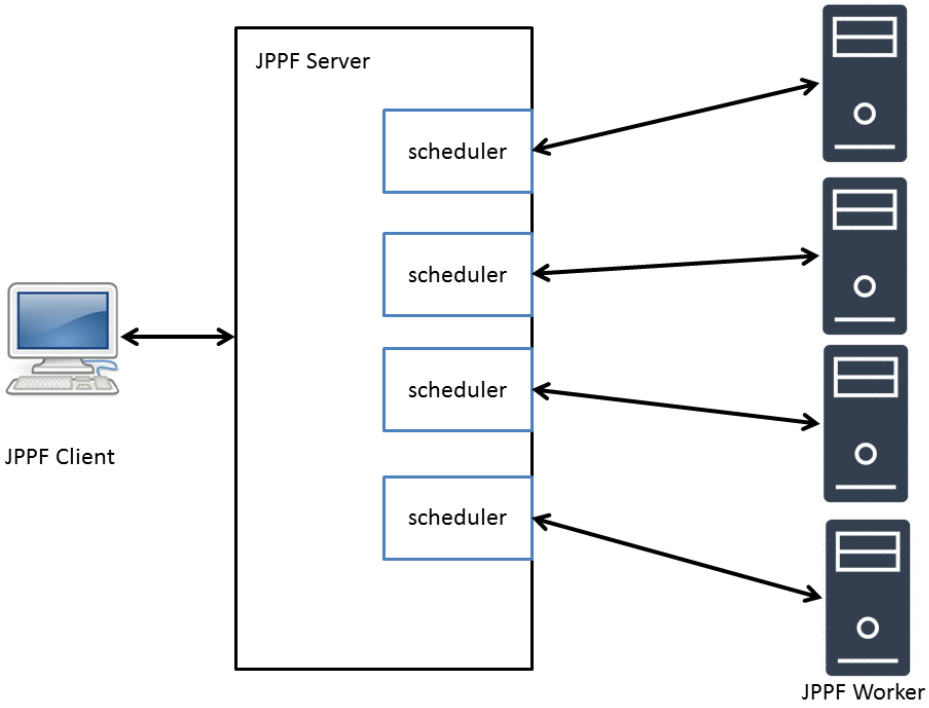
Dalam proses penjadwalan secara dinamis, tujuan dari pentingnya memiliki pengetahuan tentang penggunaan CPU suatu node adalah untuk mengukur seberapa besar suatu proses untuk dapat menggunakan sumber daya CPU pada interval waktu yang tetap (Wolski, Spring, & Hayes, 1999). Sebagai contoh, jika tersedia sumber daya CPU sebesar 50%, sebuah proses seharusnya bisa memperoleh 50% dari time-slice yang tersedia. Jika hanya 50% time-slice yang tersedia, sebuah proses diperkirakan memakan waktu 2 kali lebih lama apabila dibandingkan pada kondisi CPU tidak sedang memiliki beban komputasi yang lain. Persentase sumber daya CPU dirumuskan dengan

$$available_{cpu\_load\_average} = \left(\frac{1.0}{load_{avg} + 1.0}\right) * 100\% \qquad \textbf{(2.3)}$$

yang menunjukkan persentase waktu CPU yang tersedia untuk proses baru.

2.3. JPPF

Java Parallel Processing Framework (JPPF) adalah suatu framework yang dibangun di atas bahasa pemrograman Java. Karena dibangun di atas bahasa pemrograman Java, JPPF dapat berjalan di sistem operasi yang mendukung Java, seperti sistem operasi Windows, sistem operasi keluarga Linux, Mac OS, dan lain-lain (Cohen, 2014).



Gambar 2.2 Diagram Arsitektur JPPF

JPPF dibangun dengan arsitektur *master/slave*. Dalam JPPF seperti yang ditunjukkan pada Gambar 2.2, terdapat tiga entitas yang saling berkomunikasi:

1. *Client*, merupakan entitas yang berperan sebagai pembuat *job*. *Job* dibuat oleh *client* yang kemudian dikirim ke *server*. *Job* yang dibuat oleh *client* terdiri atas beberapa *task*.
2. *Server*, merupakan bagian yang bertugas sebagai pusat pengatur kinerja JPPF. *Server* menerima *job* dari *client* yang terdiri dari beberapa *task*. Kemudian *server* akan mendistribusikan *task-task* tersebut ke *node* yang

terpilih dengan format *bundle*. Tiap *bundle* dapat berisi *task* dalam jumlah yang berbeda. Setelah *task* selesai dikerjakan oleh *node*, server menerima hasil pekerjaan dari *node* dan meneruskan hasil tersebut ke *client*.

3. *Node*, merupakan bagian dari JPPF yang bertugas melakukan eksekusi *task* yang diberikan oleh *server*. *Node* dapat memiliki beberapa *thread* pada tiap komputer. Tiap *thread* dapat melakukan eksekusi *task* secara independen.

Selain ketiga tersebut di atas, ada beberapa istilah yang digunakan di dalam JPPF, yaitu sebagai berikut.

1. *Job*, merupakan pekerjaan yang diberikan oleh *client* kepada *server* untuk dikerjakan secara paralel. Sebuah *job* minimal memiliki sebuah *task*. *Task* dapat juga disebut dengan *subjob*.
2. *Task*, merupakan satuan unit eksekusi terkecil yang ditangani oleh JPPF. *Task* diinstansiasi oleh *client* dan membentuk grup yang disebut *job*.
3. *Bundle*, merupakan sejumlah *task* yang dikirimkan secara bersamaan oleh *server* ke suatu *node* untuk dieksekusi. Jumlah *task* dalam suatu *bundle* berubah-ubah sesuai yang ditentukan oleh *scheduler*.
4. *Feedback*, merupakan suatu rangkuman dari proses eksekusi sebuah *bundle* yang telah selesai diekseskusi untuk kemudian digunakan acuan oleh *scheduler* untuk menentukan ukuran *bundle* selanjutnya.

Penjadwalan yang dimiliki oleh JPPF menggunakan strategi manual dan dinamis. Strategi manual berjalan dengan penjadwalan statis, yaitu setiap node akan menerima sejumlah *task* sesuai yang telah ditentukan pada konfigurasi awal. Strategi manual tidak memiliki pengetahuan akan beban kerja yang ditanggung oleh suatu *node*. Setiap *node* dipandang sebagai sumber daya komputasi yang memiliki kemampuan sama.

Pada strategi dinamis, JPPF menggunakan algoritma yang terinspirasi oleh *reinforcement learning* (RL) sebagai metode untuk menyesuaikan beban yang akan diberikan kepada suatu *node*. Strategi dinamis tersebut menggunakan algoritma distribusi normal. Parameter yang digunakan untuk menentukan ukuran *bundle* adalah berasalah dari *feedback* yang berupa jumlah *task* yang telah dieksekusi dan waktu yang digunakan untuk mengerjakan *task-task* tersebut.

Gambar 2.3 menggambarkan kode sumber proses perhitungan ukuran *bundle* menggunakan algortima distribusi normal pada JPPF.

```
    @Override
    public void feedback(final int size, final double totalTime) {
        if (size <= 0) {
            return;
        }
        BundlePerformanceSample sample = new
BundlePerformanceSample(totalTime / size, size);
        dataHolder.addSample(sample);
        computeBundleSize();
    }

    protected void computeBundleSize() {
        double d = dataHolder.getPreviousMean() -
dataHolder.getMean();
        double threshold = ((RLProfile)
profile).getPerformanceVariationThreshold() *
dataHolder.getPreviousMean();
        prevBundleSize = bundleSize;
        if (action == 0) {
            action = (int) -Math.signum(d);
        }
        if ((d < -threshold) || (d > threshold)) {
            action = (int) Math.signum(action) * (int) Math.round(d
/ threshold);
        } else {
            action = 0;
        }
        int maxActionRange = ((RLProfile)
profile).getMaxActionRange();
        if (action > maxActionRange) {
            action = maxActionRange;
        } else if (action < -maxActionRange) {
            action = -maxActionRange;
        }
        bundleSize += action;
        //int max = Math.max(1, maxSize());
        int max = maxSize();
        if (bundleSize > max) {
            bundleSize = max;
        }
        if (bundleSize <= 0) {
            bundleSize = 1;
        }
    }
```

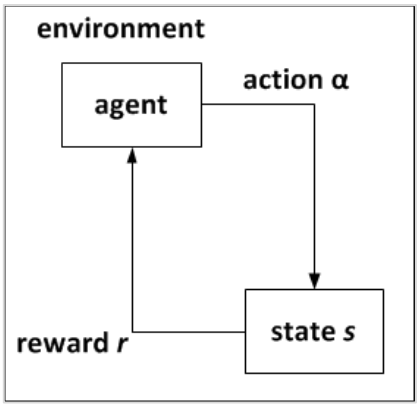
Gambar 2.3 Kode sumber pemrosesan *feedback* pada distribusi *task* menggunakan algoritma RL di dalam JPPF (Cohen, 2014)

Proses perhitungan tersebut dimulai dengan menerima *feedback* yang diterima dari *node*. *Feedback* tersebut kemudian ditambahkan ke dalam *datasample*. *Datasample* merupakan sekumpulan data *feedback* dalam jumlah tertentu yang digunakan untuk menghitung nilai *mean*. Proses perhitungan dilanjutkan dengan menghitung perbedaan *mean* yang sekarang dengan *mean*

yang sebelumnya dan menghitung nilai *threshold*. Strategi dinamis menghitung jumlah *task* yang akan dikirimkan berdasarkan perubahan nilai *mean* yang terjadi.

2.4. Reinforcement Learning

Reinforcement Learning (RL) berawal dari suatu gagasan sederhana tentang suatu sistem yang menginginkan sesuatu, yang menyesuaikan tingkah lakunya untuk memaksimalkan suatu sinyal khusus dari *environment*-nya. Tidak seperti pada bentuk *machine learning* kebanyakan, RL tidak diberikan informasi *action* apa yang harus dilakukan, tetapi harus menemukan *action* apa yang menghasilkan *reward* terbanyak dengan cara mencobanya. Dalam kasus yang paling menarik dan paling menantang adalah *action* yang dilakukan mungkin saja tidak hanya mempengaruhi *immediate reward*, tetapi juga mempengaruhi situasi selanjutnya dan *reward-reward* selanjutnya (Sutton & Barto, 1998). Secara sederhana, RL diilustrasikan pada Gambar 2.4.



Gambar 2.4 *Framework reinforcement learning*

Supervised learing bukan merupakan RL, karena *environment*, dalam kasus ini diistilahkan dengan "guru," memberikan indikasi *action* apa yang harus diambil dan secara implisit memberikan *reward* kepada bagian tertentu dalam pengambilan keputusan internal (Sutton R. S., 1984).

Sebagai contoh, kerumitan dalam sistem pembelajaran yang dihadapi dalam permainan catur. Perpindahan sangat bergantung kepada banyaknya pengambilan keputusan internal yang dilakukan oleh sistem. RL melakukan evaluasi terhadap hasil dari *action* yang dilakukan, RL tidak memberikan indikasi *action* yang harus

dilakukan dan tidak memberikan informasi sama sekali tentang proses pengambilan keputusan internal.

Satu tantangan yang muncul di dalam RL dan tidak di sistem pembelajaran yang lain adalah adanya *tradeoff* antara eksplorasi dan eksplotasi. Untuk mendapatkan *reward* yang banyak, agen RL harus memilih *action* yang pernah diambil dan secara efektif menghasilkan *reward*. Tetapi, untuk menemukan *action* seperti itu, agen RL harus mencoba *action* yang belum pernah diambil sebelumnya. Agen RL harus mengeksploitasi apa yang telah diketahuinya agar memperoleh *reward*, tetapi agen RL juga harus melakukan eksplorasi agar dapat membuat pemilihan *action* yang lebih baik di masa depan. Agen RL harus mencoba berbagai macam *action* dan secara bertahap cenderung untuk memilih *action-action* terbaik. Pada kasus *stochastic task*, tiap *action* harus dicoba berulang kali agar dapat mengestimasi *reward* secara tepat.

Di samping agen dan *environment*, di dalam RL terdapat empat sub-elemen: *policy, reward function, value function,* dan *model* dari *environment*. *Model* di dalam RL merupakan opsional.

1. *Policy*

Policy mendefinisikan bagaimana cara agen bertindak pada suatu waktu tertentu. Secara kasar, *policy* adalah sebuah pemetaan dari *state* yang diterima dari *environment* ke *action* yang akan diambil. Hal ini menyerupai dengan yang di dalam dunia psikologi disebut dengan seperangkat aturan atau asosiasi stimulus-respon. Pada beberapa kasus, *policy* dapat berupa *function* sederhana atau *lookup table*, sedangkan di kasus lain mungkin dapat berupa komputasi ekstensif seperti proses pencarian. *Policy* merupakan inti dari agen RL, dalam arti bahwa *policy* saja sudah cukup untuk menentukan *behavior*. Secara umum, *policy* bisa bersifat *stochastic*.

Salah satu metode yang digunakan agen untuk bertindak adalah dengan menggunakan algoritma ε-*greedy*. Algoritma ε-*greedy* berjalan dengan cara memilih *action* yang memiliki *value* paling tinggi pada suatu *state* tertentu. Namun, algoritma ε-*greedy* juga memiliki kemungkinan untuk memilih *action* secara acak yang ditentukan oleh nilai ε. Algoritma ini ditunjukkan pada Gambar 2.5.

```
if random < epsilon
      choose random action
else
      choose action with best value
```

Gambar 2.5 Pseudocode algoritma ε-greedy

2. *Reward function*

Reward function mendefinisikan *goal* dalam permasalahan yang dihadapi oleh RL. Dengan kata lain, *reward function* memetakan *state* (atau sepasang *state-action*) yang diterima dari *environment* menjadi suatu angka, yang menunjukan *intrinsic desirability* dari *state* tersebut. Tujuan utama agen RL adalah untuk memaksimalkan total *reward* yang diterima dalam jangka panjang. *Reward function* mendefinisikan apa yang baik dan apa yang buruk bagi agen. Namun, di dalam dunia biologi, hal ini kurang sesuai untuk mengidentifikasi *reward* dengan senang dan sakit. Tetapi, *reward* dapat dijadikan dasar untuk mengubah *policy*. Sebagai contoh, jika sebuah *action* yang dipilih oleh *policy* diikuti dengan *reward* yang rendah, kemudian *policy* mungkin diubah untuk memilih beberapa *action* lain dalam situasi tersebut di masa depan. Secara umum, *reward function* bisa juga bersifat *stochastic*.

Penggunaan *reward* merupakan fitur yang paling khas dari RL. Sebagai contoh, untuk membuat robot belajar untuk berjalan, peneliti telah menyediakan *reward* pada tiap langkah yang diambil oleh robot yang proporsional terhadap perpindahan robot. Untuk membuat robot belajar keluar dari labirin, *reward* yang diberikan adalah nol untuk tiap langkah hingga robot itu keluar dan +1 apabila robot itu berhasil keluar. Suatu pendekatan lain adalah dengan memberikan *reward* -1 untuk tiap langkah yang diambil untuk mencari jalan keluar; hal ini mendorong robot agar sesegera mungkin keluar dari labirin. Untuk membuat robot belajar untuk mencari dan mengumpulkan kaleng bekas minuman, robot menerima *reward* nol untuk sebagian besar waktu, *reward* +1 apabila robot menemukan kaleng bekas dan kaleng itu telah kosong, dan *reward* -1 apabila robot itu menabrak sesuatu.

Dari contoh di atas, agen RL selalu belajar untuk memaksimalkan *reward*. Jika ingin robot agar melakukan sesuatu, *reward* harus disediakan untuk robot

dengan cara tersebut sehingga agen memaksimalkan *reward* dan juga mencapai *goal* yang diberikan.

3. *Value function*

Berbeda dengan *reward function* yang mengindikasikan apa yang baik dari penginderaan langsung, *value function* menjelaskan apa yang baik dalam jangka panjang. Secara kasar, *value* dari sebuah *state* adalah jumlah *reward* yang dapat diharapkan oleh sebuah agen yang terakumulasi ke masa depan dari *state* tersebut. *Value* mengindikasikan *long-term desirability* dari *state* dengan mempertimbangkan *state-state* yang memiliki kemungkinan akan mengikuti, dan *reward* yang tersedia dari *state-state* tersebut. Sebagai contoh, sebuah *state* mungkin selalu menghasilkan *reward* langsung yang rendah, tetapi tetap memiliki *value* yang tinggi karena *state* tersebut biasanya diikuti oleh *state-state* lain yang menghasilkan *reward* langsung yang tinggi. Jika dianalogikan dengan manusia, *reward* diibaratkan seperti kesenangan (*reward* tinggi) dan sakit (*reward* rendah), sedangkan *value* merupakan pertimbangan yang lebih halus dan jauh ke depan tentang seberapa menyenangkan atau tidak menyenangkan ketika *environment* kita berada pada suatu *state* tertentu.

Tidak akan ada *value* jika tidak ada *reward*, dan tujuan satu-satunya menghitung *value* adalah untuk mendapatkan *reward* lebih. *Value* merupakan hal yang paling diperhatikan ketika membuat dan mengevaluasi pengambilan keputusan. Pemilihan *action* dibuat berdasarkan penilaian *value*. Agen mencari *action-action* yang menghasilkan *value* tertinggi, bukan *reward* tertinggi, karena *action-action* tersebut menghasilkan jumlah *reward* tertinggi dalam jangka panjang. Namun, menentukan nilai *value* lebih sulit daripada menentukan nilai *reward*. *Reward* diberikan secara langsung oleh *environment*, tetapi *value* harus dihitung dan dihitung kembali berdasarkan serangkaian pengamatan yang dilakukan agen selama agen tersebut melakukan interaksi. Komponen terpenting di dalam semua algoritma RL adalah metode untuk menghitung *value* secara efisien.

Salah satu metode yang digunakan untuk menghitung *value* adalah metode *Q-learning* yang dirumuskan dengan

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \quad \textbf{(2.4)}$$

dengan α adalah *learning rate* dan γ adalah *discount factor*, R_{t+1} adalah *reward* yang diamati setelah mengambil *action* A_t di dalam *state* S_t.

Learning rate menentukan seberapa besar informasi yang baru diterima akan menggantikan informasi yang lama. *Discount factor* menyatakan tingkat pentingnya informasi yang baru saja diterima.

4. *Model*

Elemen ke empat dan terakhir dari RL sistem adalah *model* dari *environment*. *Model* adalah sesuatu yang menirukan perilaku *environment*. Sebagai contoh, diberikan suatu *state* dan *action*, *model* bisa memprediksi hasil untuk *state* selanjutnya dan *reward* selanjutnya. *Model* digunakan untuk perencanaan, yaitu segala cara yang digunakan untuk memutuskan dari sebuah *action* dengan memperhatikan situasi yang mungkin terjadi di masa depan sebelum hal itu dapat dialami. Penggunaan *model* dan perencanaan di dalam RL merupakan suatu hal yang relatif baru. RL pada awalnya adalah benar-benar pembelajaran dari *trial-and-error*. Namun, perlahan metode RL sangat berhubungan erat dengan metode *dynamic programming* (DP), yaitu menggunakan model, dan juga berhubungan erat dengan metode *state-space planning*.

Beberapa permasalahan yang RL terbukti memberikan hasil yang lebih baik dibandingkan dengan *supervised learning* adalah sebagai berikut.

2.4.1. *N-armed Bandit*

Permasalahan *n-armed bandit* diilustrasikan sebagai berikut. Sebuah agen harus memilih sebuah/beberapa pilihan dari *n* pilihan tersedia. Setiap pilihan yang dipilih, agen mendapatkan *reward* dalam bentuk angka dari distribusi probabilitas stasioner yang bergantung pada *action* (pilihan) yang dipilih. Tugas agen tersebut adalah memaksimalkan total *reward* pada suatu periode tertentu, misalkan untuk 1000 pilihan *action*. Dalam permasalahan ini, setiap *action* yang dipilih disebut *play*.

Setiap pemilihan *action* adalah seperti memainkan tuas dari salah satu mesin permainan, dan *reward* adalah hadiah jika mendapatkan *jackpot*. Dengan bermain

berulang-ulang, agen berusaha memaksimalkan kemenangan dengan berkonsentrasi pada tuas terbaik. Setiap *action* memiliki *reward* yang diharapkan atau *reward* rata-rata yang diberikan jika *action* tersebut diambil. Jika agen mempertahankan estimasi nilai untuk *action-action*, maka pada tiap *play* akan terdapat minimal satu *action* yang memiliki nilai estimasi tertinggi.

```
    private void exploit() {
        double[] probSum = new double[bandit.getNumArms()];
        int sum = sumRewards();
        if (sum <= 0) { //prevent division by zero
            explore();
            return;
        }
        probSum[0] = rewards[0] / (double) sum;
        for (int i = 1; i < probSum.length; ++i) {
            probSum[i] = probSum[i - 1] + rewards[i] /
(double) sum;
        }
        double d = rand.nextDouble();
        int n = 0;
        while (n < probSum.length && d > probSum[n]) {
            ++n;
        }
        rewards[n] += bandit.play(n);
    }
```

Gambar 2.6 Contoh kode sumber proses eksploitasi pada permasalahan *n-armed bandit* (Rehnel, 2013)

```
    private void explore() {
        int n = rand.nextInt(bandit.getNumArms());
        rewards[n] += bandit.play(n);
    }
```

Gambar 2.7 Contoh kode sumber proses eksplorasi pada permasalahan *n-armed bandit* (Rehnel, 2013)

Metode ini disebut *greedy action*. Dengan melakukan *greedy action*, agen dikatakan melakukan eksploitasi pengetahuan yang dimiliki terhadap *value* dari *action-action* yang pernah diambil seperti yang ditunjukkan pada Gambar 2.6. Jika agen melakukan *non-greedy action*, maka agen melakukan eksplorasi untuk meningkatkan estimasi *value* dari *action-action* yang lain seperti yang ditunjukkan pada Gambar 2.7. Melakukan eksploitasi merupakan tindakan yang tepat untuk memaksimalkan *reward* dalam suatu *play*, namun eksplorasi mungkin menghasilkan menghasilkan total *reward* yang lebih besar dalam jangka panjang. Sebagai contoh, misalkan *value* dari *greedy action* diketahui, sedangkan beberapa *action* lain diestimasi memiliki *value* yang hamper sama namun dengan

ketidakpastian yang cukup besar. Ketidakpastian dari *action* lain tersebut mungkin lebih baik dari *greedy action*. Jika agen masih memiliki *play* yang cukup banyak tersisa, maka sebaiknya agen melakukan eksplorasi terhadap *non-greedy action* dan menemukan *action* mana yang lebih baik daripada *greedy action*. *Reward* dalam jangka pendek yang diperoleh dari eksplorasi mungkin saja lebih sedikit, tetapi bisa saja menghasilkan total *reward* yang lebih besar dan kemudian dieksploitasi.

Dengan $Q^*(a)$ sebagai *value* sebenarnya dari *action* $\boldsymbol{a}$ dan *value* yang diestimasi setelah $\boldsymbol{t}$ *play* dengan $Q_t(a)$. *Value* sebenarnya dari sebuah *action* adalah *reward* rata-rata yang diperoleh jika *action* tersebut dipilih. Salah satu cara alami untuk menghitung nilai estimasi adalah dengan menghitung rata-rata dari *reward-reward* yang diterima ketika *action* tersebut dipilih. Setelah $\boldsymbol{t}$ *play*, *action* $\boldsymbol{a}$ telah dipilih sebanyak k_a kali dan menghasilkan *reward* $r_1, r_2, \dots, r_{k_a}$, sehingga *value* dari *action* diestimasi dengan:

$$Q_t(a) = \frac{r_1 + r_2 + \cdots + r_{k_a}}{k_a} \tag{2.5}$$

jika $k_a = 0$, maka $Q_t(a)$ didenisikan ke suatu nilai tertentu, misal 0. Ketika $k_a \to \infty$, maka nilai $Q_t(a)$ akan mengerucut ke $Q^*(a)$. Metode ini disebut metode *sample-average* untuk menghitung estimasi *value* dari *action*.

Aturan pemilihan *action* yang paling sederhana adalah memilih *action* dengan *value* estimasi tertinggi, misalkan untuk memilih *greedy-action* a_t^* pada *play* ke-t dengan $Q_{t-1}(a_t^*) = max_a Q_{t-1}(a)$. Metode ini selalu mengeksploitasi pengetahuan saat ini untuk memaksimalkan *reward* langsung. Alternative sederhana adalah untuk berperilaku *greedy* pada sebagian besar waktu, tetapi pada suatu waktu, dengan probabilitas kecil ε, memilih *action* secara acak. Metode ini disebut ε-*greedy*. Keuntungan dari metode ini adalah dengan semakin banyak *play*, akan diambil *sample* dari tiap *action*, dan bahwa semua $Q_t(a)$ mengerucut ke $Q^*(a)$. Hal ini menunjukkan bahwa probabilitas untuk memilih *action* yang optimal adalah menuju $1 - \varepsilon$.

2.4.2. Cart-Pole Balancing

Tugas agen RL dalam permasalah *cart-pole balancing* adalah memberikan gaya kepada troli (*cart)* agar bergerak di suatu lintasan untuk menjaga tongkat (*pole*) tegak di atas troli agar tidak terjatuh. *Failure* terjadi ketika tongkat terjatuh melewati sudut tertentu terhadap posisi vertikal atau troli melebihi lintasan yang telah ditentukan. Dalam permasalahan ini, tugas agen RL dapat dikategorikan sebagai *episodic* atapun *continual*.

Tugas agen dikatakan sebagai *episodic* apabila ada batas alami berupa *final step*, yaitu ketika interaksi antara agen dan *environment* terhenti. Tiap berhentinya interaksi antara agen dan *environment* disebut *episode*. Setiap *episode* berakhir pada suatu *state* yang disebut *terminal state*, diikuti dengan diatur-ulangnya keadaan ke *state* mula-mula. Sedangkan, pada banyak kasus, interaksi antara agen dan *environment* tidak terhenti oleh *episode* yang dapat teridentifikasi, tetapi interaksi tersebut terus berlangsung tidak berhingga.

Tugas agen RL dalam permasalah *cart-pole balancing* dapat dikategorikan ke dalam *episodic* karena pada tiap terjadi *failure* (tongkat terjatuh), *environment* dikembalikan ke keadaan mula-mula. *Reward* yang diberikan adalah +1 untuk semua *action* yang tidak menghasilkan *failure*. Sedangkan, tugas agen dapat dikategorikan ke dalam *continual* dengan memberikan *reward* -1 ketika terjadi *failure* dan *reward* 0 untuk *action-action* yang tidak menghasilkan *failure*.

2.5. *Markov Property*

Menurut Sutton (1998), di dalam *framework* RL, agen membuat keputusan dengan berdasarkan sinyal yang berasal dari *environment* yang disebut dengan *state*. Yang disebut dengan *state* tersebut adalah segala informasi yang tersedia bagi agen. Sinyal *state* dapat berupa pembacaan langsung dari sensor-sensor ataupun hasil pembacaan yang telah diproses lebih lanjut. Secara ideal, sinyal *state* adalah suatu nilai yang merangkum pembacaan-pembacaan sensor secara padat namun tetap relevan terhadap informasi yang diperoleh. Pada umumnya, sinyal *state* membutuhkan lebih dari pembacaan langsung, namun tidak pernah lebih dari riwayat lengkap dari semua pembacaan langsung. Sebuah sinyal *state* yang berhasil dalam menjaga semua informasi yang relevan disebut memiliki

Markov property. Sebagai contoh, posisi bidak pada permainan catur merangkum semua transisi penting yang mengarah pada posisi saat ini. Banyak informasi tentang urutan transisi telah hilang, tetapi hal itu tidak terlalu dibutuhkan karena seluruh hal yang berkaitan dengan kelanjutan permainan tersebut tetap terjaga.

Untuk lebih sederhana secara matematika, dalam *Markov property* diasumsikan memiliki jumlah *state* dan *value* yang terbatas. Dengan memperhatikan bagaimana *environment* secara umum memberikan respon yang mungkin diberikan pada waktu $t + 1$ terhadap *action* yang diambil pada waktu t. Pada sebagian besar kasus, respon yang diberikan bisa jadi bergantung kepada seluruh yang terjadi sebelumnya. Dalam kasus ini hanya dapat dijelaskan dengan mendefinisikan distribusi probabilitas secara lengkap:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \quad \textbf{(2.6)}$$

untuk semua r, s', dan semua kemungkinan nilai dari peristiwa yang telah lalu: $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$. Jika *state* tersebut memiliki *Markov property*, maka respon dari *environment* pada $t + 1$ bergantung hanya pada *state* dan *action* pada t, yang dapat dituliskan dengan:

$$Pr\{R_{t+1} = r, S_{t+1} = s' | S_t, A_t\} \quad \textbf{(2.7)}$$

untuk semua $r, s', S_t,$ dan A_t. Dengan kata lain, sebuah sinyal *state* memiliki *Markov property* dan merupakan *Markov state*, jika dan hanya jika (2.6) sama dengan (2.7).

Jika *environment* memiliki *Markov property*, maka hal ini satu langkah lebih dinamis yang memungkin untuk memprediksi *state* selanjutnya dan *reward* selanjutnya yang mungkin didapatkan jika agen mengambil *action* pada *state* saat ini. Agen dapat memprediksi semua kemungkinan *state* di masa depan dan *reward* yang diharapkan dari pengetahuan hanya dari *state* saat ini sebagaimana baiknya jika agen memperoleh riwayat lengkap hingga saat ini.

Markov property menjadi sangat penting di dalam RL karena pengambilan keputusan dan *value* dapat diasumsikan sebagai fungsi hanya dari *state* saat ini. Representasi *state* harus informatif agar pengambilan keputusan dan perhitungan *value* menjadi efektif dan informatif.

```
int _state = 0;

if (_cart_position < -2.4
        || _cart_position > 2.4
        || _pole_angle < -twelve_degrees
        || _pole_angle > twelve_degrees) {
    return (-1);
    /* to signal failure */
}

if (_cart_position < -0.8) {
    _state = 0;
} else if (_cart_position < 0.8) {
    _state = 1;
} else {
    _state = 2;
}

if (_cart_velocity < -0.5) {
} else if (_cart_velocity < 0.5) {
    _state += 3;
} else {
    _state += 6;
}

if (_pole_angle < -six_degrees) {
} else if (_pole_angle < -one_degree) {
    _state += 9;
} else if (_pole_angle < 0) {
    _state += 18;
} else if (_pole_angle < one_degree) {
    _state += 27;
} else if (_pole_angle < six_degrees) {
    _state += 36;
} else {
    _state += 45;
}

if (_pole_angular_velocity < -fifty_degrees) {
} else if (_pole_angular_velocity < fifty_degrees) {
    _state += 54;
} else {
    _state += 108;
}
```

Gambar 2.8 Proses pengolahan sinyal *state* menjadi *Markov state* pada permasalahan *cart-pole balancing* (Sutton R. S., 1984)

Di dalam permasalah *cart-pole balancing* pada 2.4.2, sinyal *state* akan menjadi *Markov* jika sinyal tersebut dituliskan secara tepat, atau dibuat agar sinyal tersebut memungkinkan untuk direkonstruksi secara tepat, posisi dan kecepatan troli pada lintasan, sudut antara troli dan tongkat, dan kecepatan perubahan sudut (kecepatan sudut/angular). Di dalam sistem yang ideal, informasi ini akan cukup

untuk memprediksi secara tepat perilaku troli dan tongkat di masa depan dan *action* yang diambil oleh sistem. Di dalam praktik, hal ini tidak pernah mungkin untuk mengetahui informasi secara tepat karena tiap sensor akan mengalami distorsi dan *delay* dalam melakukan pengukuran.

Contoh pemrosesan sinyal *state* pada permasalahan *cart-pole balancing* agar menjadi *Markov* adalah seperti yang ditunjukkan pada Gambar 2.8 yang merupakan disertasi dari Richard S. Sutton. Pertama, dilakukan pengecekan terhadap posisi troli dan sudut tongkat terhadap troli. *State* dikatakan *failure* jika posisi troli atau sudut tongkat terhadap troli berada di luar lintasan atau melebihi sudut yang diizinkan. Jika terjadi *failure*, maka *Markov state* adalah -1. Jika sinyal *state* dinyatakan bukan *failure*, maka terdapat kemungkin 162 *Markov state* yang ditentukan berdasarkan posisi troli, kecepatan troli, sudut tongkat dan kecepatan sudut tongkat terhadap troli. Dari keempat sinyal *state* dapat direpresentasikan oleh satu *Markov state* secara tepat dan dapat direkontruksi kembali menjadi empat sinyal *state*.

RL yang memenuhi kriteria *Markov property* disebut *Markov decision process* (MDP). Jika ruang *state* dan *action* adalah terbatas, maka disebut *finite MDP*. *Finite MDP* didefinisikan oleh himpunan *state* dan *action*, dan oleh prediksi keadaan *environment*. Diketahui *state* s dan *action* a, probabilitas dari kemungkinan tiap *state* selanjutnya s' adalah

$$p(s'|s,a) = \Pr\{S_{t+1} = s'|S_t = s, A_t = a\}. \quad \textbf{(2.8)}$$

Persamaan (2.8) disebut *transition probabilities*. Hampir sama dengan sebelumnya, diketahui *state* s dan *action* a, nilai *reward* yang diharapkan dari *state* selanjutnya adalah

$$r(s,a,s') = \mathbb{E}[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s']. \quad \textbf{(2.9)}$$

BAB III
METODOLOGI PENELITIAN

Pada Bab III ini akan dijabarkan mengenai metodologi penelitian yang diusulkan. Penelitian ini akan dibagi dalam enam tahapan, diantaranya studi literatur, perancangan penelitian, implementasi penelitian, uji coba dan analisis penelitian serta penulisan laporan dan paper.

3.1. Perumusan Masalah

Masalah yang dihadapi dalam penelitian ini adalah perbaikan metode *load balancing* untuk komputasi klaster dengan bantuan *Java Parallel Processing Framework* (JPPF) pada kondisi dinamis. JPPF memiliki metode pendistribusian *task* yang bersifat statis dan dinamis. Metode statis mendistribusikan *task* dalam jumlah yang tetap ke tiap *node* yang tergabung dalam sistem. Metode dinamis mendistribusikan *task* ke tiap *node* yang tergabung dalam sistem dengan ukuran yang fleksibel yang mengikuti algoritma tertentu. Metode dinamis cocok digunakan dalam kondisi lingkungan yang dinamis.

Pada JPPF, implementasi metode distribusi dinamis menggunakan algoritma distribusi normal. Metode dinamis berjalan dengan membandingkan rata-rata waktu eksekusi per *task* pada sebuah sampel sebesar *n*. Rata-rata waktu eksekusi diperoleh dari *feedback* yang diberikan oleh sistem setiap kali sebuah *bundle* yang terdiri dari beberapa *task* selesai dieksekusi. Jika setelah *feedback* ditambahkan ke dalam sampel dan nilai *mean* berubah melebihi nilai suatu *threshold* yang sudah ditentukan, maka metode dinamis menghitung perubahan jumlah *task* yang besarnya berbanding lurus terhadap perubahan *mean*.

Namun, metode dinamis memiliki kekurangan, yaitu kondisi lingkungan yang sebenarnya mungkin saja berbeda dengan kondisi lingkungan yang diasumsikan oleh pembuat metode dinamis. Agar pendistribusian *task* tetap optimal meskipun berada dalam lingkungan dinamis yang tidak seperti yang diasumsikan dalam metode dinamis, maka dibutuhkan metode distribusi adaptif yang menawarkan kemampuan mampu mempelajari dan menyesuaikan dengan kondisi lingkungan untuk mengoptimalkan pembagian porsi pendistribusian.

Hal tersebut mendasari adanya perbaikan mekanisme *load balancing* untuk komputasi klaster dengan *reinforcement learning* (RL) pada kondisi dinamis. Penulis berpendapat, metode adaptif dengan menggunakan RL dapat meningkatkan performa distribusi *task* pada kondisi dinamis. Untuk menguji hipotesa pada penelitian ini, dilakukan beberapa persiapan lingkungan pengujian, seperti lingkungan komputasi, mengintegrasikan algoritma RL ke dalam JPPF dan melakukan pengujian performa.

3.2. Studi Literatur

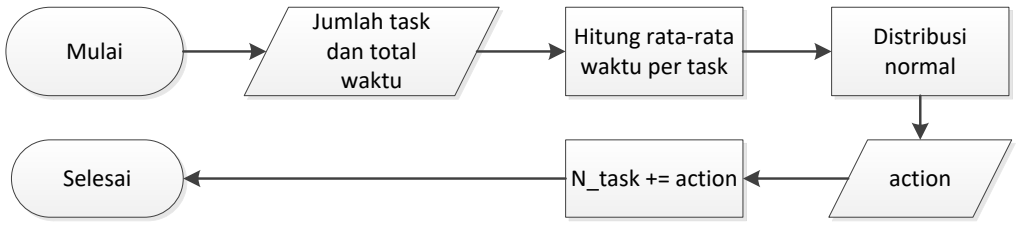
Tahapan ini merupakan tahapan pencarian referensi pembelajaran yang dapat dipertanggungjawabkan seperti referensi yang bersumber dari buku, jurnal atau artikel. Pencarian referensi berguna untuk menambah wawasan dari penelitian terdahulu yang terkait sebagai penunjang untuk membuat suatu penelitian yang dapat bermanfaat di kemudian hari. Topik-topik referensi yang terkait dengan penelitian ini diantaranya mengenai komputasi klaster, struktur JPPF, penjadwalan *job* yang pernah diteliti, matriks-matriks yang berpengaruh terhadap perhitungan penjadwalan dan algoritma-algoritma yang digunakan di dalam *reinforcement learning* (RL) beserta beberapa permasalahan yang mengimplementasikan RL. Pendefinisian masalah menjadi lebih jelas dan detil dengan mendalami topik-topik terkait penelitian.

3.3. Desain Sistem

Arsitektur sistem komputasi klaster yang diusulkan untuk penelitian ini secara umum ditunjukkan pada Gambar 2.1. Arsitektur yang akan dibangun terdiri atas sebuah *client*, sebuah *server*, dan empat buat *node* JPPF. *Client* berperan sebagai pembuat *job*. *Server* berperan menerima *job* yang dikirim oleh *client* dan mengirimkan *task* yang terdapat di dalam *job* ke *node* yang terpilih. *Node* berperan sebagai eksekutor *task* yang dikirim oleh *server*. *Server* JPPF adalah bagian JPPF yang bertugas menentukan jumlah *task* yang akan diberikan ke suatu *node*. *Client, server* dan tiap *node* dijalankan di atas Docker dan diatur kecepatan CPU dan *bandwidth* yang sama. Penjadwalan yang dimiliki oleh JPPF menggunakan strategi statis dan strategi dinamis.

Strategi dinamis pada JPPF menggunakan metode distribusi normal, yang oleh *author*-nya disebut sebagai metode yang terinspirasi RL. Pada JPPF, metode dinamis menerima *feedback* dari *node* berupa jumlah *task* yang dieksekusi dan total waktu yang dipakai untuk mengeksekusi *task-task* tersebut. Algoritma metode dinamis ditunjukkan pada Gambar 2.3.

Feedback yang berupa jumlah *task* dan total waktu tersebut kemudian dihitung rata-rata sehingga diperoleh waktu yang digunakan untuk mengeksekusi satu *task*. Setelah diperoleh waktu rata-rata yang digunakan untuk mengeksekusi suatu *task*, waktu rata-rata tersebut digunakan oleh metode dinamis untuk menghitung *action*. *Action* yang dimaksud adalah jumlah *task* yang akan ditambahkan atau dikurangkan untuk dieksekusi. Alur proses perhitungan jumlah *task* digambarkan pada Gambar 3.1.

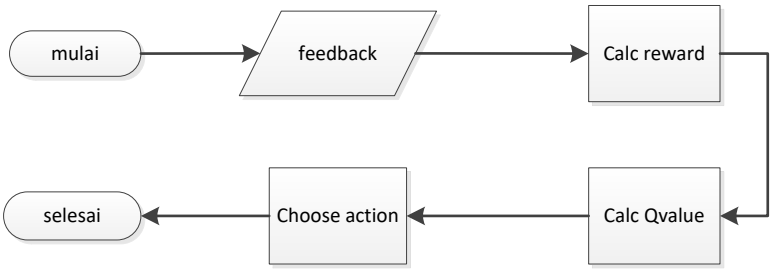


Gambar 3.1 *Flowchart* perhitungan jumlah *task* pada JPPF

Pada penelitian ini, diusulkan penggunaan algoritma RL sebagai metode adaptif untuk pendistribusian *task* dari *server* ke *node*. Metode RL yang diusulkan mengolah *feedback* berupa rata-rata waktu eksekusi per *task*, *throughput* jaringan, beban CPU dan ukuran *bundle*. Namun tidak semua *feedback* tersebut diolah dalam satu skenario tunggal, melainkan dibagi-bagi menjadi beberapa skenario. Skenario pertama adalah RL dengan parameter rata-rata waktu eksekusi per *task*. Skenario kedua adalah RL dengan parameter rata-rata waktu eksekusi per *task*, *throughput* jaringan dan beban CPU. Skenario ketiga adalah RL dengan parameter ukuran *bundle* sebelumnya. Pembagian skenario ini bertujuan untuk mengetahui jenis *feedback* yang paling baik dalam merepresentasikan *state* di dalam algoritma RL. Alur proses perhitungan jumlah *task* ditunjukkan pada Gambar 3.2.

Proses perhitungan *task* dimulai dengan menerima *feedback* dari proses eksekusi *task* sebelumnya. Dari *feedback* tersebut ditentukan *reward* untuk *action* yang sebelumnya diambil sekaligus memperbarui nilai *Q-value* untuk *state-action*

yang bersangkutan. Setelah proses penghitungan *reward* dan *Q-value* selesai, proses dilanjutkan dengan pengambilan *action* menggunakan algortima ε-*greedy*. *Action* yang diperoleh merepresentasikan besarnya perubahan jumlah *task* yang akan diberikan ke suatu *node*, dapat bernilai positif maupun negatif.



Gambar 3.2 *Flowchart* perhitungan *task* menggunakan algoritma RL

3.3.1. Metode distribusi RL dengan *feedback* rata-rata waktu eksekusi per *task*

Pada pendekatan ini, proses penghitungan jumlah *task* adalah dengan menggunakan algoritma RL dengan parameter rata-rata waktu eksekusi per *task*. Setiap kali suatu hasil eksekusi *task* diterima oleh *server* dari *node*, diperoleh juga *feedback* yang merupakan rangkuman dari proses eksekusi tersebut. Salah satu *feedback* tersebut adalah rata-rata waktu eksekusi per *task*. Data tersebut kemudian dimasukkan ke dalam sampel yang memiliki ukuran tertentu. Dari proses tersebut, dapat diperoleh *mean* dan *previous mean*. Kedua nilai tersebut kemudian digunakan untuk memperoleh nilai rasio antara *previous mean* dan *mean*. Nilai rasio tersebut kemudian diubah menjadi *state* yang digunakan untuk pengambilan *action* oleh algoritma RL. Proses penghitungan *task* dengan algoritma RL dengan parameter rata-rata waktu eksekusi ditunjukkan pada Gambar 3.3.

```
Feedback average execution time
Add feedback to sample
Calculate mean
Calculate previous mean
Ratio ← previous mean / mean
State ← convert ratio to state
Choose action from state using ε-greedy
```

Gambar 3.3 Pseudocode pengambilan *action* pada algoritma RL dengan parameter rata-rata waktu eksekusi

3.3.2. Metode distribusi RL dengan parameter rata-rata waktu eksekusi, *throughput* jaringan dan beban CPU

Pada pendekatan ini, proses penghitungan jumlah *task* adalah dengan menggunakan algoritma RL dengan parameter rata-rata waktu eksekusi per *task*, *throughput* jaringan dan beban CPU seperti yang ditunjukkan pada Gambar 3.4. Masing-masing *feedback* ditambahkan ke dalam sampel-sampel yang bersesuaian, *feedback* rata-rata waktu eksekusi ditambahkan ke dalam sampel rata-rata waktu eksekusi, *feedback throughput* ditambahkan ke dalam sampel *throughput* dan *feedback* beban CPU ditambahkan ke dalam sampel beban CPU. Dari masing-masing sampel tersebut, diambil rasio-rasio yang kemudian diubah menjadi *state*. Algoritma RL kemudian memilih *action* berdasarkan *state* yang diperoleh dari *feedback* tersebut.

```
Feedback average execution time, throughput, CPU load
Add feedback to sample
Get mean execution time
Get previous mean execution time
Get mean throughput
Get previous mean throughput
Get mean CPU load
Get previous mean CPU load
Ratio exec time ← previous mean execution time / mean execution time
Ratio throughput ← previous mean throughput / mean throughput
Ration CPU load ← previous mean CPU load / mean CPU load
State ← convert ratio exec time, ratio throughput and ratio CPU load to
     state
Choose action from state using ε-greedy
```

Gambar 3.4 Pseudocode pengambilan *action* pada algoritma RL dengan parameter rata-rata waktu eksekusi, *throughput* jaringan dan beban CPU

3.3.3. Metode distribusi RL dengan parameter ukuran *bundle*

Pada pendekatan ini, perhitungan *task* adalah dengan menggunakan algoritma RL dengan parameter ukuran *bundle*. Ukuran *bundle* dapat disebut juga dengan jumlah *task*, karena *bundle* disusun oleh *task*. Pada metode ini, rasio didapatkan dengan membandingkan ukuran *bundle* pada eksekusi sebelumnya dengan ukuran maksimal suatu *bundle*. Perhitungan jumlah *task* pada metode ini ditunjukkan pada Gambar 3.5.

```
Feedback bundle size
Ratio ← bundle size / max bundle size
State ← convert ratio to state
Choose action from state using ε-greedy
```

Gambar 3.5 Pseudocode pengambilan *action* pada algoritma RL dengan parameter ukuran *bundle*

3.4. Implementasi Penelitian

Tahapan implementasi merupakan tahapan eksekusi rancangan penelitian. Pada proses implementasi ini, perlu diketahui spesifikasi perangkat keras dan perangkat lunak yang akan digunakan. Perangkat keras yang digunakan adalah dua buah komputer yang pada masing-masing komputer dipasang *virtual machine*. Selain perangkat keras, penelitian ini juga memerlukan perangkat lunak dengan spesifikasi sebagai berikut:

- Java 7
- Netbeans 8
- JPPF versi 4.2.2
- Sistem operasi Ubuntu 16.04
- Docker versi 1.12

Adapun tahap-tahap implementasi dari rancangan penelitian dijelaskan sebagai berikut.

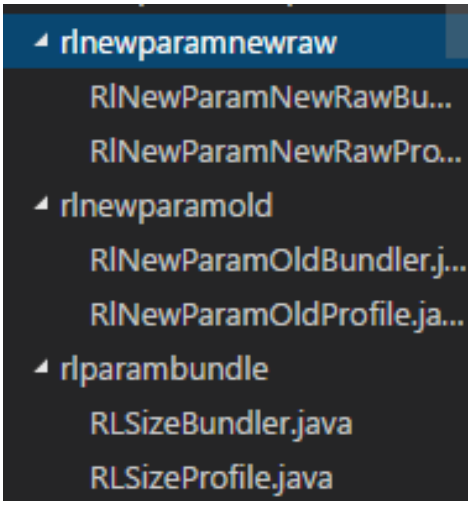
3.4.1. Pembuatan *load balancer* baru

Bagian *load balancing* di dalam JPPF berperan untuk memecah-mecah *job* menjadi *task-task*, menentukan jumlah *task* yang akan dikirimkan ke *node*. Pendistribusian *task* dilakukan pada sisi JPPF *driver* (*server*).

Tahap pertama dalam membuat *load balancer* baru adalah dengan menambahkan `package` baru ke dalam kode sumber JPPF. Dalam penelitian ini, seperti yang telah dijelaskan pada bagian desain sistem, terdapat tiga metode baru, sehingga jumlah `package` baru yang ditambahkan adalah sebanyak tiga buah. Di dalam setiap `package` terdapat dua `class`, yaitu `class bundler` dan `class profile`. `Class bundler` berperan melakukan komputasi untuk menentukan jumlah *task* yang akan diberikan ke suatu *node*. Sedangkan, `class profile` berperan untuk menyimpan konfigurasi untuk suatu metode *load*

balancing yang ditentukan oleh pengguna. Contoh implementasi penambahan `package` yang dilakukan dalam penelitian ini ditunjukkan pada Gambar 3.6.

Agar `class-class` *load balancer* baru yang telah ditambahkan dapat dikenali oleh *framework* JPPF dan melakukan proses distribusi *task*, maka diperlukan dua langkah. Langkah pertama adalah menambahkan tiga `class` baru yang mengimplementasikan `JPPFBundlerProvider`. `Class` ini berfungsi untuk memberikan nama terhadap *load balacher* yang telah dibuat dan sebagai jembatan agar *framework* JPPF dapat menginstansiasi dan menggunakan *load balancer* yang telah ditambahkan.



Gambar 3.6 *Package* RL

Langkah kedua adalah menambahkan tiga entri yang berisi *full path* dari `provider-provider` ke dalam berkas `org.jppf.server.scheduler.-bundle.spi.JPPFBundlerProvider` yang terletak di dalam folder `META-INF/services`. Hal ini bertujuan agar *load balancer* yang telah ditambahkan dapat dikenali oleh JPPF sebagai metode *load balancing* yang tersedia.

3.4.2. Implementasi metode *load balancing* menggunakan RL

Implementasi pada bagian ini adalah penerapan algoritma RL ke dalam metode *load balancing* di dalam `package` yang telah ditambahkan. Ketiga metode *load balancing* baru mengikuti prinsip yang sama, yaitu menerima

feedback, menghitung *reward*, menghitung *Q-value*, mengubah *feedback* menjadi *state* dan memilih *action*.

```
    public void feedbackMyRL(int nbTask, long elapsed, long
elapsedInNode, long transportTime, int tasksSize, int resultSize)
{
        super.feedbackMyRL(nbTask, elapsed, elapsedInNode,
transportTime, tasksSize, resultSize);
        computeBundle();
    }
    private void computeBundle() {
        epsilon = getEpsilon();
        if (policy[currentState] != null) {
            int immediateReward = computeReward();
            policy[currentState].setCummulativeReward(action,
immediateReward);
        }
        double acak = random.nextDouble();
        int max = maxSize();
        prevState = currentState;
        currentState = MyRLHelper.getState2(max, bundleSize);
        if (policy[currentState] == null) {
            policy[currentState] = new MyRLPolicy(getUUID());
        }
        if (!firstTime2) {
policy[currentState].setPreviousState(policy[prevState]);
        }
        firstTime2 = false;

        if (acak > epsilon) {
            action = policy[currentState].getExploitAction(bundleSize,
max, maxActionRange);
        } else {
            action = policy[currentState].getExploreAction(bundleSize,
max, maxActionRange);
        }

        bundleSize += action;
        if (bundleSize > max) {
            bundleSize = max;
        }
        if (bundleSize <= 0) {
            bundleSize = 1;
        }
    }
```

Gambar 3.7 Kode sumber implementasi algoritma RL dengan *feedback* ukuran *bundle*

Salah satu implementasi adalah algoritma RL dengan parameter ukuran *bundle* yang ditunjukkan pada Gambar 3.7. Algoritma RL dimulai dengan menerima *feedback*, namun pada skenario ini, hanya *feedback* ukuran *bundle* yang digunakan untuk diubah menjadi *state*. *Feedback* ukuran *bundle* tersebut direpresentasikan oleh variabel `nbTask`. Langkah selanjutnya adalah menghitung *reward*, memperbarui *Q-value*, dan memilih *action* berdasarkan *state* yang diperoleh menggunakan algoritma ε-*greedy*.

3.4.3. Implementasi perhitungan *reward*

Implementasi pada bagian ini adalah penerapan perhitungan *reward* yang akan diberikan kepada suatu *pair* dari *state-action*. Suatu *action* diberikan *reward* positif jika total waktu yang digunakan untuk mengeksekusi suatu *bundle* di suatu *node* lebih kecil dibandingkan dengan rata-rata waktu global yang digunakan untuk mengeksekusi. Sebaliknya, *action* akan diberikan *reward* negatif jika total waktu yang digunakan untuk mengeksekusi suatu *bundle* di suatu *node* lebih lama dibandingkan dengan rata-rata waktu global yang digunakan untuk mengeksekusi. Implementasi metode perhitungan *reward* ini ditunjukkan pada Gambar 3.8.

```
    protected int computeReward() {
        int count = 0;
        double mean2 = 0;
        for (AbstractTesisBundler b : bundlers) {
            BundleDataHolder h = b.getExecHolder();
            mean2 += b.getTotalTime();
            count++;
        }
        double globalMean = mean2 / count;

        int reward = 0;
        if (totalTime <= globalMean) {
            reward = 3;
            if (prevNbTask > nbTask) {
                reward = 1;
            }
        }
        else {
            reward = 0 - (int) ((totalTime / globalMean));
        }
        lastReward = reward;
        return reward;
    }
```

Gambar 3.8 Kode sumber implementasi perhitungan *reward* untuk algoritma RL

Pada kode sumber perhitungan *reward* yang ditunjukkan pada Gambar 3.8, langkah pertama yang dilakukan adalah menjumlah tiap-tiap waktu eksekusi eksekusi yang disimpan oleh *bundler*. Setiap *node* yang terhubung ke *server*, JPPF membuatkan *bundler* tersendiri untuk *node* tersebut. *Bundler* merupakan algoritma *load balancing* itu sendiri, yang juga menyimpan informasi-informasi dinamis yang berkaitan dengan kondisi suatu *node*.

Setelah total waktu eksekusi dari setiap *bundler* diperoleh, maka dihitung nilai rata-rata waktu eksekusi global. Variabel `totalTime` merupakan waktu yang digunakan oleh *node* ini sendiri, sehingga jika waktu eksekusi suatu *node*

lebih kecil daripada rata-rata global, maka diberikan *reward* +3. Namun, jika waktu eksekusi suatu *node* lebih kecil daripada rata-rata global tetapi ukuran *bundle* yang sekarang lebih kecil dibandingkan dengan ukuran *bundle* pada *episode* sebelumnya, maka *reward* yang diberikan adalah +1.

Jika waktu eksekusi *suatu* node lebih besar daripada rata-rata waktu eksekusi global, maka diberikan *reward* negatif yang besarnya seperti yang ditunjukkan pada Gambar 3.8.

3.4.4. Implemetasi metode *Q-learning* untuk memperbarui nilai *Q-value*

Pada setiap *reward* yang diberikan kepada *state-action* tertentu, perlu dilakukan pembaruan terhadap *value* untuk *state-action* tersebut. *Value* sebuah *action a* pada suatu *state s* menunjukkan seberapa bagus *action a* tersebut untuk diambil pada *state s*. Secara sederhana, *value* dari sebuah *action* pada suatu *state* adalah akumulasi *reward* yang diperoleh.

Salah satu metode yang digunakan untuk mengolah *reward* agar menjadi *value* adalah *Q-learning*. *Value* yang diperoleh dari hasil perhitungan menggunakan *Q-learning* disebut dengan *Q-value*. Metode ini ditunjukkan pada Gambar 3.9.

```
    private void calculateQValue(double futureStateQValue) {
//        Q[s,a] ←(1-α) Q[s,a] + α(r + γmaxa' Q[s',a']).
        double immediateReward = 0;
        if (immediateRewardMap.containsKey(currentAction)) {
            immediateReward =
immediateRewardMap.get(currentAction);
        }
        double qValue = getQValue(currentAction);
        qValue = ((1 - getStepSize()) * qValue) + (getStepSize() *
(immediateReward + (getDiscount() * futureStateQValue)));
        qValueMap.put(currentAction, qValue);    }
```

Gambar 3.9 Implementasi metode *Q-learning* untuk memperbarui *Q-value*

3.4.5. Implementasi penentuan *state* untuk algoritma RL

Algoritma RL berjalan dengan dengan mengenali sebuah *state* kemudian mengambil *action* dan menyimpan *reward* yang diperoleh. Dikarenakan *feedback* yang diperoleh dari JPPF belum berbentuk dalam *state* yang dapat dimengerti oleh agen RL, maka diperlukan metode untuk mengubah *feedback* menjadi *state* yang dapat dimengerti oleh agen RL.

Dalam penelitian ini, diusulkan tiga jenis metode *load balancing* yang menggunakan algoritma RL sebagai agen pembelajaran, yaitu metode *load balancing* dengan parameter rata-rata waktu eksekusi, metode *load balancing* dengan parameter rata-rata waktu eksekusi, *throughput* jaringan dan beban CPU, dan metode *load balancing* dengan parameter jumlah *task*. Sehingga, diimplementasikan cara mengubah *feedback* menjadi *state* RL. Gambar 3.10 menunjukkan algoritma untuk mengubah *feedback* berupa waktu eksekusi, *throughput* jaringan dan beban CPU menjadi *state* yang dapat dipahami oleh agen RL. Sedangkan, Gambar 3.11 menunjukkan algoritma untuk mengubah *feedback* berupa jumlah *task* menjadi *state* yang dapat dimengerti oleh agen RL.

```
ratio = prevMean / mean
RATIO ← array [0.5, 0.8, 1, 1.2]
length ← 4
for i=0 to length do
    if ratio < RATIO[i]
        return i
    done
end
return length
```

Gambar 3.10 *Pseudocode* untuk mengubah *feedback* waktu eksekusi, *throughput* jaringan dan beban CPU menjadi *state* RL

```
nTask <-- current number of task
total_task_in_job <-- number of task in one job

state <-- 10 * nTask / total_task_in_job

return state
```

Gambar 3.11 *Pseudocode* untuk mengubah *feedback* jumlah *task* menjadi *state* RL

3.4.6. Implementasi metode ε-*greedy* untuk memilih

Setiap kali suatu *bundle* selesai dieksekusi, maka metode *load balancing* akan menentukan besarnya perubahan ukuran *bundle* selanjutnya yang diberikan ke suatu *node*. Besarnya perubahan ukuran *bundle* tersebut direpresentasikan oleh *action*. *Action* yang dipilih adalah *action* yang terbaik pada suatu *state* tertentu. Namun, apabila suatu *state* tidak memiliki pengetahuan tentang *action*, maka diberikan *action* yang dipilih secara acak. Metode yang digunakan untuk memilih *action* ditunjukkan pada Gambar 3.12.

```
    public int getExploitAction(int currentBundleSize, int maxBundleSize,
int maxActionRange) {
        double bestValue = -99999999;
        int selectedAction = -1;
        numOfBestAction = -1;
        Set<Map.Entry<Integer, Double>> qValues = qValueMap.entrySet();
        for (Map.Entry<Integer, Double> map1 : qValues) {
            int action = map1.getKey();
            int bundleSize = currentBundleSize + action;
            double value = map1.getValue();
            if (bundleSize > 0 && bundleSize <= maxBundleSize)
            {
                if (value > bestValue) {
                    bestValue = value;
                    selectedAction = action;
                    numOfBestAction = 0;
                    listBestAction[numOfBestAction] = selectedAction;
                } else if (value == bestValue) {
                    numOfBestAction++;
                    listBestAction[numOfBestAction] = action;
                }
            }
        }
        if (numOfBestAction < 0) {
            return getExploreAction(currentBundleSize, maxBundleSize,
maxActionRange);
        }
        if (numOfBestAction > 0) {
            int index = (int) (Math.random() * (numOfBestAction + 1));
            selectedAction = listBestAction[index];
        }
        return selectedAction;
    }
    public int getExploreAction(int currentBundleSize, int maxBundleSize,
int maxActionRange) {
        int bundlesize = 0;
        int action = 0;
        while (bundlesize < 1 || bundlesize > maxBundleSize) {
            action = (int) (Math.random() * maxActionRange);
            double isPositif = Math.random() * 10;
            if (isPositif < 5) {
                action *= -1;
            }
            bundlesize = currentBundleSize + action;
        }
        return action;
    }
```

Gambar 3.12 Implementasi metode ε-*greedy* untuk memilih *action*

Seperti yang ditunjukkan pada Gambar 3.12, pemilih *action* terbaik dengan mencari *action* yang memiliki *Q-value* tertinggi disebut dengan proses eksploitasi. Sedangkan, jika pada proses eksploitasi tidak ditemukan *action*, maka proses pemilihan *action* dilakukan secara acak yang disebut juga dengan proses eksplorasi. Proses eksplorasi juga dapat dilakukan oleh algoritma RL jika proses agen RL memutuskan untuk melakukan *action* ekplorasi untuk memilih *action* alternatif selain *action-action* yang memiliki *Q-value* terbaik.

3.4.7. Implementasi perubahan *throughput* jaringan

Implementasi perubahan *throughput* bertujuan untuk memberikan perubahan *throughput* jaringan antara suatu *node* dan *server*. Untuk melakukan hal ini, digunakan perintah `tc qdisc` yang berjalan di sistem operasi keluarga Linux. Panjang perintah `tc qdisc` cukup panjag, sehingga untuk memudahkan penggunaan, diimplementasikan ke dalam suatu *script* yang dapat dieksekusi melalui *terminal* Linux. Implementasi *script* untuk mengubah *throughput* jaringan ditunjukkan pada Gambar 3.13. *Script* ini dijalankan selama 60 detik dan ditiadakan selama 60 detik secara berulang-ulang. Sehingga, selama proses eksekusi *job*, terdapat kondisi *throughput* yang berubah-ubah.

```
#!/bin/sh

if [ "$1" = "" ]; then
    echo "harap masukkan bandwidth dalam kbps"
    exit
fi

echo "change bandwidth to $1"

#delete all network config on eth 0
tc qdisc del dev eth0 root

#from http://lartc.org/howto/lartc.qdisc.classless.html
tc qdisc add dev eth0 root tbf rate "$1"Mbit latency 250ms burst 1540

echo "done"
```

Gambar 3.13 Implementasi *script* untuk mengubah *throughput* jaringan

3.4.8. Implementasi perubahan beban CPU

Implementasi perubahan beban CPU betujuan untuk memberikan perubahan beban CPU pada suatu *node*. Untuk melakukan hal ini, sebuah *task* dekripsi MD5 dijalankan untuk memecahkan *hashed text* menjadi *plaintext*. *Task* dibuat secara sederhana dan dapat diselesaikan dengan waktu eksekusi selama kurang lebih 5 detik pada suatu *node*. Eksekusi *task* tersebut dilakukan selama 12 kali, sehingga total waktu eksekusi *task* adalah kurang lebih selama 60 detik. Eksekusi *task* juga ditiadakan selama 60 detik. Proses ini dilakukan berulang-ulang selama proses eksekusi *job* dari *server* berlangsung. Pemberian beban kepada CPU ini diimplementasikan dalam bentuk *script* yang ditunjukkan pada Gambar 3.14.

```
#!/bin/sh

tidur=60
while [ 1 ]
do
    l=12
    for i in $(seq 1 $l)
    do
        java -cp dist/jppf4.2.2-md5.jar tes.Tes -plain ab
        echo "\n\n"
    done
    echo "sleep selama $tidur detik"
    sleep "$tidur"
done
```

Gambar 3.14 Implementasi *script* untuk mengubah beban CPU

3.4.9. Implementasi Docker

Pada penelitian ini, Docker berperan sebagai *container* untuk menjalakan *client*, *server* dan *node*. Docker memberikan kemampuan untuk mengisolasi program untuk hanya menggunakan inti CPU tertentu. Docker juga memberikan kemampuan untuk membatasi prosentase penggunaan sumber daya komputasi, sehingga suatu *container* hanya dapat memberikan beban komputasi pada CPU sesuai dengan yang telah didefinsikan pada saat instansiasi. Untuk menjalankan *client*, *server* maupun *node* diimplementasikan ke dalam bentuk *script* yang ditunjukkan pada Gambar 3.15.

```
#!/bin/bash

if [ "$1" = "" ]; then
    echo "provide host directory for app binary source, exmp = worker1"
    exit
fi

if [ "$2" = "" ]; then
    echo "provide cpu core to host the process, exmp = 3"
    exit
fi

docker run -it --rm --name "$1" --cap-add=NET_ADMIN --cpuset-cpus="$2" --
cpu-period=50000 --cpu-quota=25000 "$network" -v
/home/zar/docker_share/"$1":/home myubuntu2
```

Gambar 3.15 Implementasi *script* Docker untuk menjalankan *container*

3.5. Pengujian dan Evaluasi

Pengujian dilakukan untuk mengetahui seberapa besar presentase peningkatan kinerja JPPF. Presentase peningkatan kinerja diketahui dengan membandingkan kinerja dari JPPF menggunakan strategi adaptif lama dengan kinerja JPPF dengan metode yang diusulkan. Parameter yang akan diuji meliputi:

- rata-rata waktu penyelesaian job pada kondisi normal;

- rata-rata waktu penyelesaian job saat salah satu atau beberapa node memiliki kondisi yang dinamis, antara lain: *throughput* yang berubah dan memiliki beban kerja yang tinggi;
- dan, kinerja mekanisme *load-balancing* menggunakan metode dinamis dengan algoritma distribusi normal dan metode adaptif menggunakan algoritma RL dengan tambahan parameter yang diusulkan ketika salah satu node mengalami kondisi yang dinamis.

Adapun evaluasi pengujian bertujuan untuk mendapatkan data empiris dari hasil uji coba dengan skenario yang telah ditetapkan dengan parameter uji coba yang meliputi rata-rata waktu eksekusi *task* di *node*, *throughput* jaringan dan beban CPU.

[Halman ini sengaja dikosongkan]

BAB IV
HASIL DAN PEMBAHASAN

Tahap pengujian dilakukan untuk menguji kebenaran dari hipotesa serta menjawab pertanyaan penelitian yang diuraikan pada BAB II.

4.1. Tahapan Penelitian

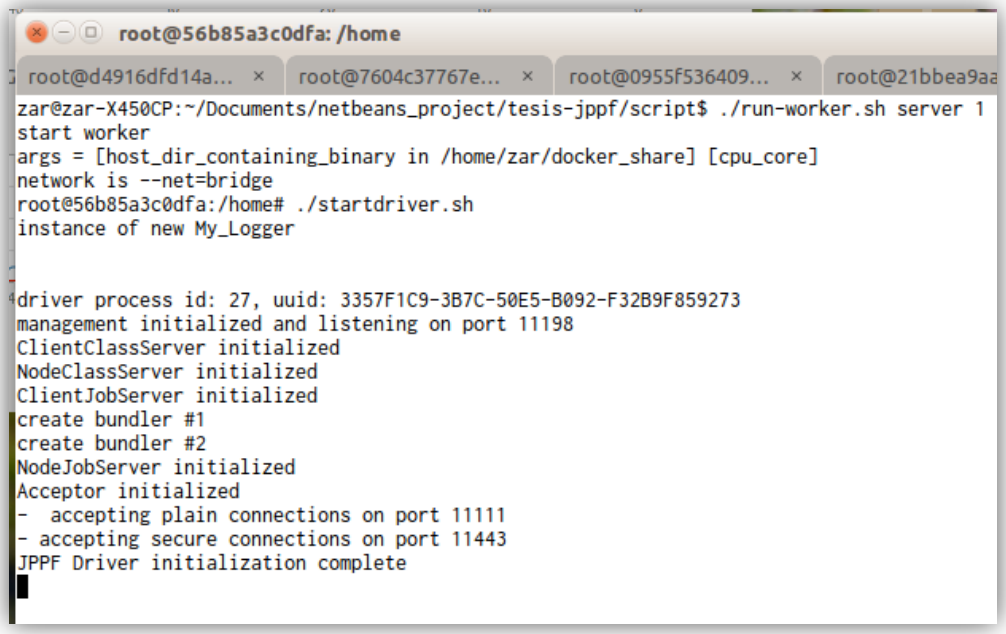
Penelitian ini dilakukan di lingkungan uji coba yang nyata. Secara umum, penelitian perbaikan mekanisme *load balancing* pada komputasi klaster dilakukan melalui beberapa tahapan sebagai berikut:

1. Tahap pertama, merupakan tahap perancangan dan implementasi sistem yang diawali dengan menambahkan modul *load balancing* baru ke dalam *framework* JPPF yang berperan sebagai pendistribusi *task* kepada *node*.
2. Tahap kedua, merupakan tahapan perancangan skenario uji coba yang meliputi skenario uji coba fungsionalitas dan skenario uji coba performa. Skenario pengujian fungsionalitas adalah fungsionalitas untuk melakukan distribusi *task* ke *node*. Sedangkan skenario pengujian performa adalah performa *load balancing* ketika sistem mengalami kondisi dinamis, seperti *throughput* jaringan yang berubah dan CPU mendapat beban komputasi dari proses lain.
3. Tahap ketiga, merupakan tahap pengujian dan perbandingan metode yang diusulkan dengan metode konvensional.

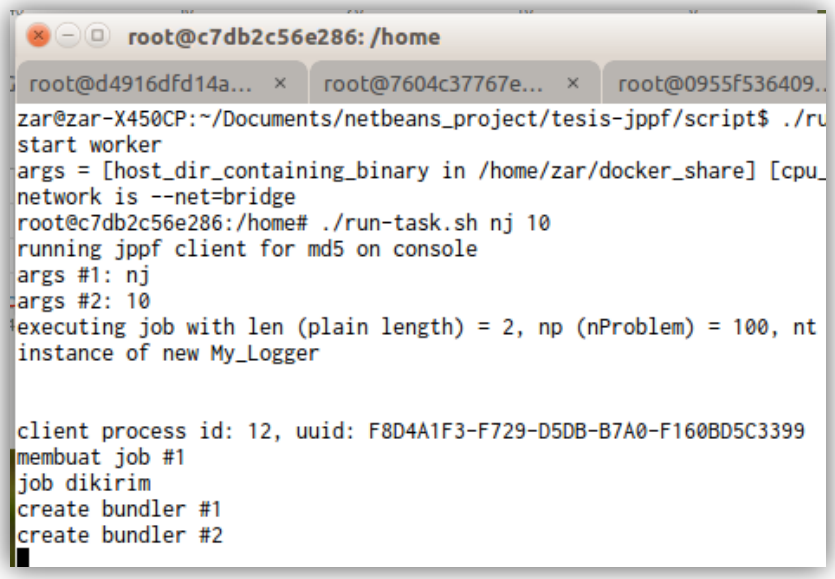
4.2. Implementasi sistem

Pada tahapan ini, sistem diimplementasikan dalam kondisi yang sebenarnya. Sistem diimplementasikan ke dalam dua bagian, yaitu bagian yang bertugas pendistribusian *task* dan aplikasi *client*, sedangkan pada bagian *node* tidak mengalami perubahan karena berperan sebagai eksekutor *task* yang diberikan. Pembagian ke dalam dua bagian ini bertujuan agar sistem dapat berjalan sesuai dengan keadaan sebenarnya. Gambar 4.1 dan Gambar 4.2 merupakan gambar antarmuka hasil *screenshot* dari sistem yang sudah diimplementasikan. Gambar 4.1 merupakan tampilan dari JPPF *server* yang berperan menampilkan status *server*. Gambar 4.2 merupakan aplikasi *client* yang akan membuat *job* untuk dikerjakan secara paralel melalui perantara JPPF *server*.

Pengguna menentukan jumlah *job* yang dikerjakan oleh sistem komputasi klaster dengan menambahkan parameter `nj` dengan diikuti nilai jumlah *job*. Setiap *job* berisi 100 *task* yang bertugas mencari *plaintext* dari 100 *hashedtext* MD5.



Gambar 4.1 Antarmuka JPPF *server*



Gambar 4.2 Antarmuka aplikasi *client*

4.3. Uji Coba

Tahapan uji coba pada penelitian ini meliputi penentuan skenario pengujian, variabel pengujian, lingkungan pengujian dan perbandingan dengan metode konvensional.

4.3.1. Lingkungan Uji Coba

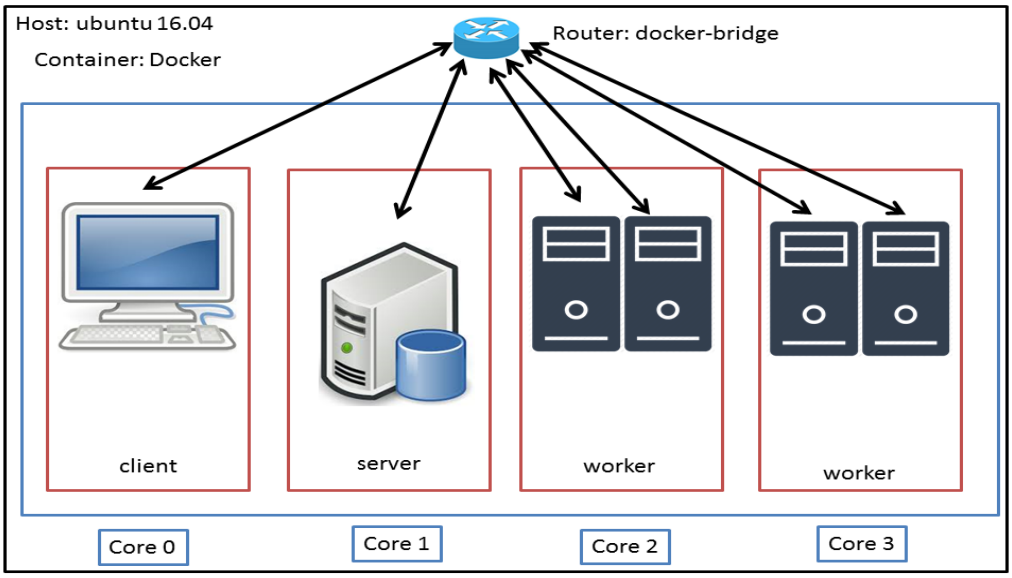
Pengujian dilakukan menggunakan sebuah komputer dengan memanfaatkan teknologi *container* yang berjalan di sistem operasi Linux. Adapun spesifikasi dari komputer yang digunakan sebagai pengujian ditunjukkan pada Tabel 4.1.

Tabel 4.1 Spesifikasi komputer pengujian

CPU	Intel Core i3 1.8GHz
RAM	6GB DDR3
Sistem operasi	Ubuntu 16.04
Container	Docker versi 1.12

Pengujian dilakukan menggunakan komputer yang menjadi *host* untuk enam elemen, yang terdiri dari sebuah *client*, sebuah *server* dan empat buah *node*. *Client* dijalankan untuk menggunakan sumber daya komputasi hanya pada inti CPU pertama, *server* dijalankan untuk menggunakan sumber daya komputasi hanya pada inti CPU kedua. Sedangkan, untuk *node*, inti CPU ketiga dan inti CPU keempat digunakan untuk menjalankan keempat *node*. *Node1* dan *node2* berjalan di inti CPU ketiga dan *node3* dan *node4* berjalan di inti CPU keempat.

Masing-masing dari keenam elemen mendapatkan komputasi sebesar 50% dari kapasitas tiap inti CPU. Semua elemen saling terhubung satu sama lain melalui *bridge* yang telah disediakan oleh Docker. Tiap elemen menjalankan *base image* Ubuntu. Konfigurasi elemen JPPF terhadap inti CPU ditunjukkan pada Gambar 4.3.



Gambar 4.3 Topologi jaringan pada pengujian

4.3.2. Skenario Pengujian

Skenario pengujian ini dilakukan untuk mengetahui hasil dari penerapan algoritma RL sebagai *load balancer* pada JPPF. Serta untuk mengetahui pengaruh algoritma RL kepada mekanisme *load balancing*.

Tabel 4.2 Tabel skenario pengujian

Skenario	Keterangan
Skenario 1	Keempat *node* memiliki *throughput* sebesar 10Mbps dan beban CPU sebesar 0%
Skenario 2	Salah satu *node* mengalami *throughput* yang berubah-ubah antara 2Mbps dan 10Mbps, serta beban CPU sebesar 0% dan 50% yang diberikan selama satu menit secara bergantian.

Skenario pengujian yang dilakukan pada penelitian ini ditunjukkan pada Tabel 4.2. Terdapat dua skenario yang digunakan untuk pengujian, yaitu Skenario 1 dan Skenario 2. Pada Skenario 1, semua *node* diatur memiliki *throughput* yang sama selama proses uji coba, yaitu sebesar 10Mbps dan sumber CPU sepenuhnya digunakan untuk eksekusi *task*.

Pada Skenario 2, salah satu *node* diatur memiliki *throughput* yang berubah-ubah selama proses eksekusi *task*, yaitu sebesar 2Mbps dan 10Mbps.

Masing-masing *throughput* diberlakukan selama satu menit secara bergantian. Sumber daya CPU juga dikenai suatu beban komputasi sebesar 0% dan 50% yang masing-masing diberlakukan selama satu menit secara bergantian.

Kasus uji yang digunakan sebagai proses pengujian adalah proses pemecahan *cipheredtext* MD5 menjadi *plaintext*. Proses ini dilakukan dengan melakukan *bruteforce* terhadap kemungkinan *plaintext* yang telah ditentukan batasannya. Dalam pengujian ini, disediakan 100 *cipheredtext* dari algoritma MD5 yang disusun dari dua karakter *plaintext*. Tiap satuan *task* akan mendapatkan porsi daerah pencarian kemungkinan *plaintext* yang dibagi menurut jumlah total *task*. Jumlah *job* yang digunakan sebagai pengujian adalah sebanyak 25.000 *job*, dengan masing-masing *job* berisi 100 *task*.

Tabel 4.3 Tabel kasus uji kinerja mekanisme *load balancing*

Pengujian	Keterangan
LBD	Pengujian kinerja mekanisme *load balancing* dinamis menggunakan algoritma distribusi normal dengan acuan rata-rata waktu eksekusi per *task*. Mekanisme *load balancing* dinamis digunakan sebagai pembanding untuk kinerja mekanisme *load balancing* yang diusulkan
LBA 1	Pengujian kinerja mekanisme *load balancing* adaptif menggunakan algoritma RL dengan parameter rata-rata waktu eksekusi per *task*
LBA 2	Pengujian kinerja mekanisme *load balancing* adaptif menggunakan algoritma RL dengan parameter rata-rata waktu eksekusi per *task*, rata-rata *throughput* jaringan dan rata-rata beban CPU
LBA 3	Pengujian kinerja mekanisme *load balancing* adaptif menggunakan algoritma RL dengan parameter ukuran *bundle*

Pada masing-masing skenario pengujian, dilakukan empat pengujian yang terdiri dari LBD, LBA 1, LBA 2 dan LBA 3 seperti yang dijelaskan pada Tabel 4.3. Pengujian ini dilakukan untuk mengetahui kinerja masing-masing metode *load balancing* ketika dijalankan pada Skenario 1 dan Skenario 2.

4.3.3. Parameter Pengujian

Adapun parameter pengujian yang digunakan pada penelitian ini agar dapat dijadikan referensi pembanding untuk metode yang diusulkan adalah sebagai berikut:

1. Rata-rata waktu eksekusi per *job*

Pengujian ini bertujuan untuk mengetahui rata-rata waktu yang diperlukan untuk menyelesaikan suatu *job* pada masing-masing skenario pengujian. Pengujian dilakukan dengan memberikan 25000 *job* untuk dikerjakan oleh sistem. Pada pengujian ini, metode *load balancing* yang baik yaitu yang mampu memberikan waktu eksekusi per job yang paling minimum di berbagai skenario pengujian.

2. Rata-rata distribusi *task* dari *server* ke *node*

Pengujian ini bertujuan untuk mengetahui kinerja metode *load balancing* terhadap pembagian porsi *task* dari *server* kepada *node*. Pengujian dilakukan dengan memberikan 25000 *job* untuk dikerjakan oleh sistem. Pada pengujian ini, metode *load balancing* yang baik yaitu metode yang mampu mengalihkan sebagian porsi *task* dari *node* yang memiliki waktu penyelesaian yang lebih besar kepada *node* yang memiliki waktu penyelesaian yang lebih kecil.

3. Total waktu yang digunakan untuk menyelesaikan seluruh *job*

Pengujian ini bertujuan untuk mengetahui total waktu yang digunakan untuk menyelesaikan seluruh *job* pada masing-masing skenario atau disebut dengan *makespan*. Pada pengujian ini, metode *load balancing* yang baik yaitu metode yang mampu memberikan total waktu yang lebih kecil. *Makespan* dari tiap skenario dibandingkan untuk mendapatkan *speedup*.

4.4. Hasil Uji Coba dan Analisis

Tujuan dari pengujian ini adalah untuk mendapatkan data empiris hasil implementasi algoritma RL sebagai metode *load balancing.* Pengujian dilakukan pada dua skenario, yaitu Skenario 1 dan Skenario 2 seperti yang dijelaskan pada Tabel 4.2.

Pada Skenario 1, keempat *node* yang berperan sebagai sumber daya yang berperan sebagai eksekutor *task,* diatur untuk memiliki *throughput* tetap selama

proses uji coba, yaitu sebesar 10Mbps. Selain itu, sumber daya CPU juga diatur untuk tidak dikenai beban komputasi selain *task* yang diberikan oleh *server*.

Sedangkan, pada Skenario 2, salah satu *node* diatur untuk mengalami perubahan *throughput* selama proses eksekusi berlangsung. Perubahan *throughput* tersebut bernilai 2Mbps dan 10Mbps yang masing-masing diterapkan selama satu menit secara bergantian. Selain itu, sumber daya CPU juga diatur untuk mengalami perubahan selama proses komputasi berlangsung, yaitu diatur untuk memiliki beban 0% dan 50% yang masing-masing diterapkan kurang lebih selama satu menit secara bergantian.

Masing-masing skenario diuji dengan metode *load balancing* seperti yang ditunjukkan pada Tabel 4.3, yaitu LBD, LBA 1, LBA 2 dan LBA 3. Pada LBD, metode *load balancing* lama menggunakan algoritma distribusi normal dengan acuan rata-rata waktu eksekusi per *task*. Metode *load balancing* menggunakan algoritma distribusi normal adalah metode yang secara *default* tersedia pada JPPF.

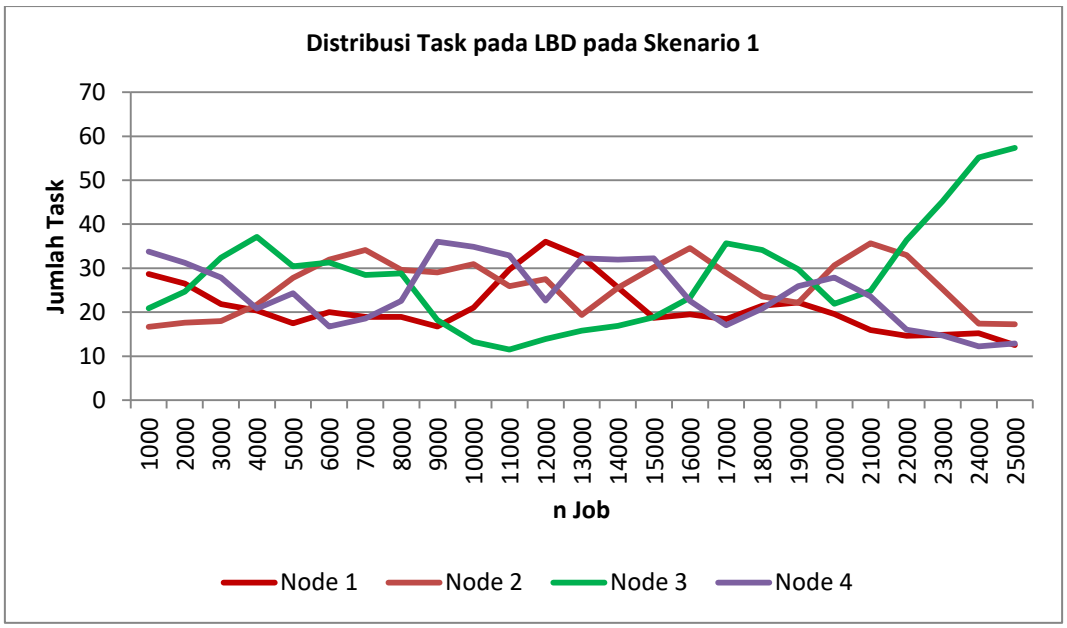
Metode usulan pada penelitian ini direpresentasikan oleh LBA 1, LBA 2 dan LBA 3. Masing-masing kasus uji tersebut menggunakan algoritma RL namun dengan parameter yang berbeda. LBA 1 merupakan metode *load balancing* menggunakan algoritma RL dengan parameter rata-rata waktu eksekusi per *task*. LBA 2 merupakan metode *load balancing* menggunakan algoritma RL dengan parameter rata-rata waktu eksekusi per *task*, rata-rata *throughput* jaringan dan rata-rata beban CPU. Sedangkang, LBA 3 merupakan metode *load balancing* menggunakan algoritma RL dengan parameter ukuran *bundle*. *Bundle* merupakan paket yang berisi satu atau beberapa *task* yang dikirim oleh *server* ke *node*.

Setiap kasus uji, diberikan *job* sejumlah 25000 *job*. Masing-masing *job* berisi 100 *task* dan 100 *cipheredtext* MD5. 100 *task* tersebut akan didistribusikan ke *node-node* oleh *load balancer* untuk menemukan *plaintext*.

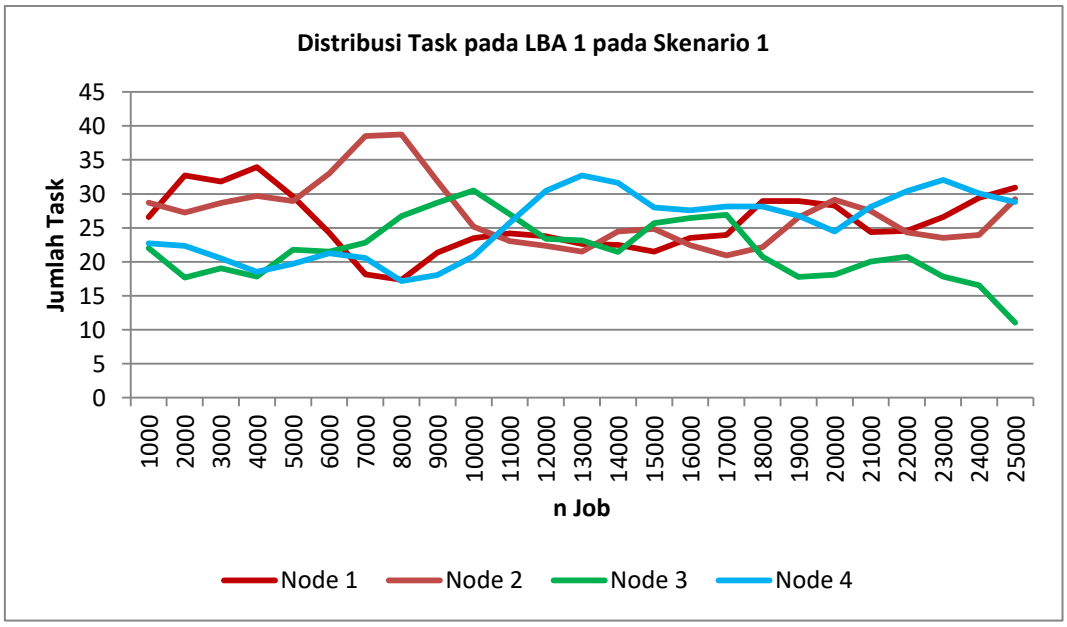
4.4.1. Hasil pengujian Skenario 1

Pada pengujian Skenario 1, *node-node* diatur untuk memiliki *throughput* jaringan sebesar 10Mbps dan beban CPU sebesar 0%. Sumber daya komputasi dan jaringan difokuskan untuk melakukan eksekusi *task* yang diberikan oleh *server*. Uji coba dilakukan dengan memberikan 25000 *job* kepada sistem dengan pengujian LBD, LBA 1, LBA 2 dan LBA 3.

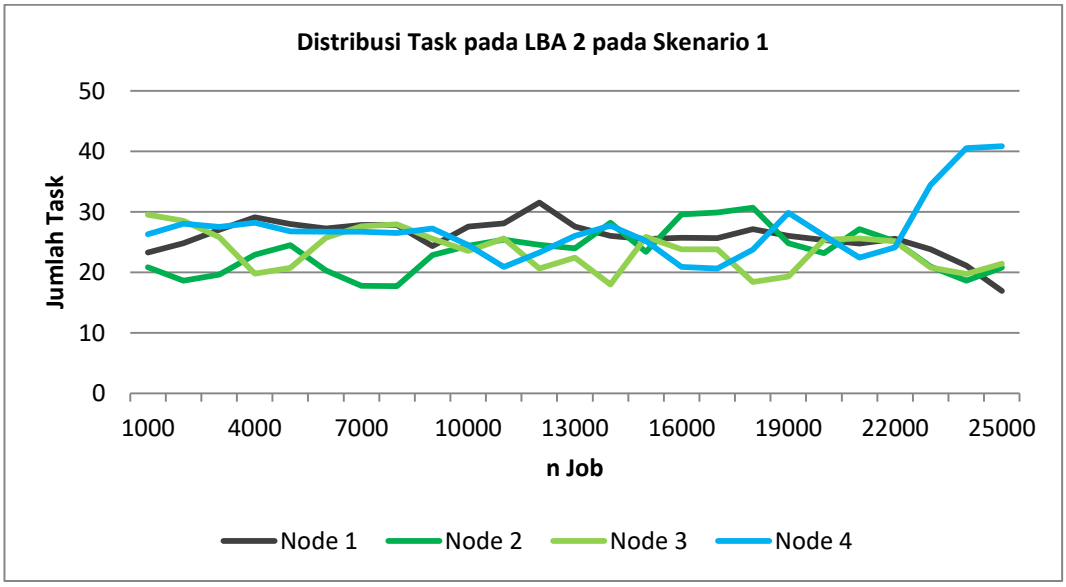
Gambar 4.4 menunjukkan hasil pengujian ditinjau dari distribusi jumlah *task* ke masing-masing *node*. Hasil dari LBD, LBA 1 dan LBA 2 menunjukkan terjadinya fluktuasi jumlah *task* yang cukup besar yang diberikan pada setiap iterasi *job* yang dieksekusi. Fluktuasi terjadi meskipun kondisi jaringan sudah diatur memiliki *throughput* yang konstan sebesar 10Mbps dan beban CPU sebesar 0%.



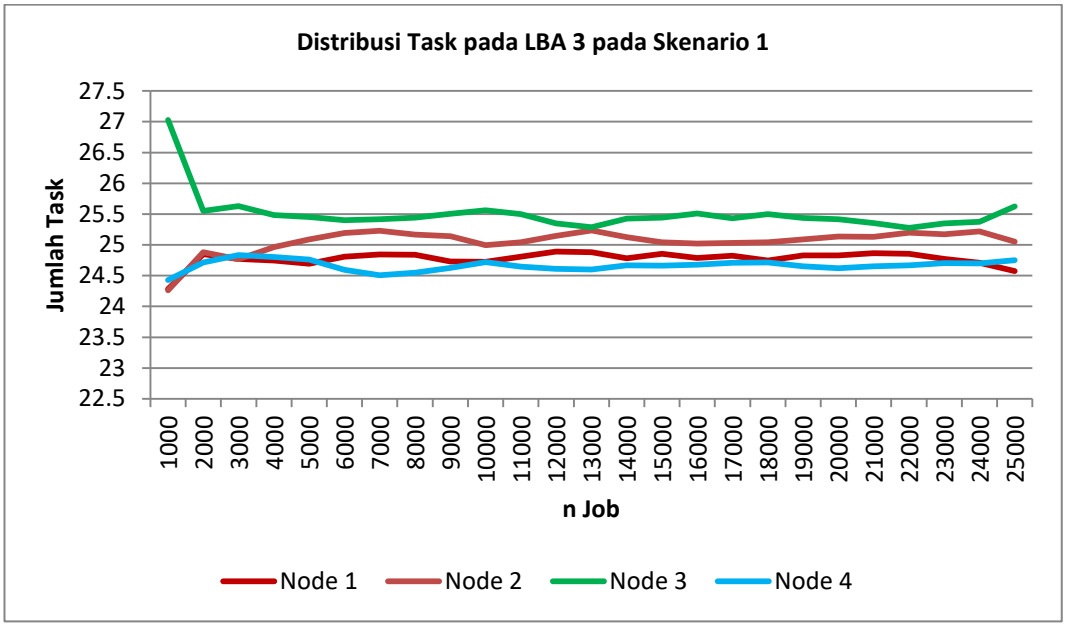
(a)



(b)



(c)



(d)

Gambar 4.4 Grafik distribusi *task* pada Skenario 1

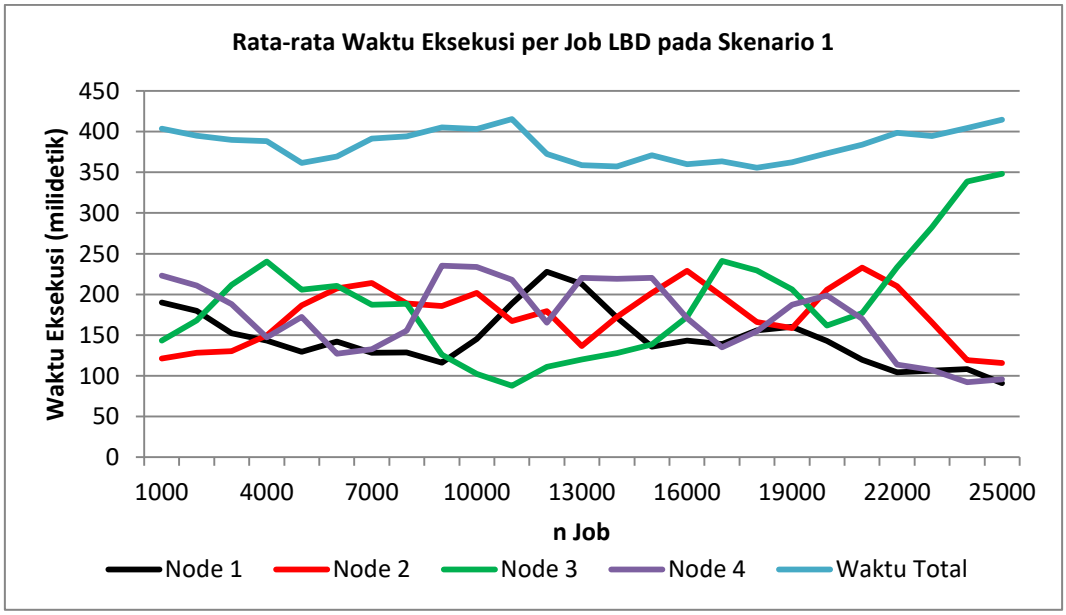
Pada hasil LBD pada Skenario 1, di awal-awal iterasi, jumlah *task* yang diberikan kepada *node 3* adalah sekitar 20 *task*, kemudian menjadi 37 *task* pada iterasi ke-4000, kemudian turun lagi menjadi 11 *task* pada iterasi ke-11000 dan naik lagi menjadi 55 *task* pada iterasi ke-24000.

Pada hasil LBA 1 pada Skenario 1, di awal-awal iterasi, jumlah *task* yang diberikan kepada *node 3* adalah sekitar 23 *task*, kemudian meningkat menjadi 30 *task* pada iterasi ke-10000 dan turun menjadi 10 *task* pada iterasi ke-25000.

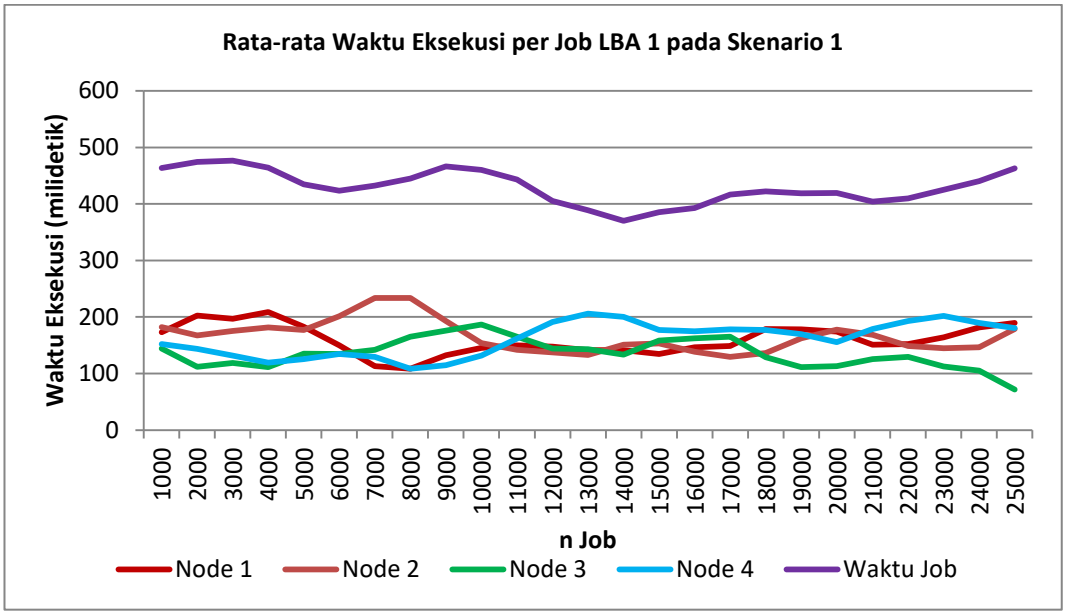
Pada hasil LBA 2 pada Skenario 1, di awal-awal iterasi, jumlah *task* yang diberikan kepada masing-masing *node* berubah-ubah antara 20 *task* hingga 30 *task*. Namun, pada akhir-akhir iterasi, *node 4* mengalami lonjakan jumlah *task* yang diterima dari sekitar 25 *task* menjadi 40 *task*.

Hasil berbeda ditunjukkan pada hasil LBA 3 pada Skenario 1. Fluktuasi jumlah *task* yang didistribusikan ke masing-masing *node* cenderung kecil dan stabil. Hanya pada awal-awal iterasi terjadi fluktuasi yang cukup signifikan, yaitu jumlah *task* yang diberikan kepada *node 3* berkurang dari angka 27 *task* ke kisaran 25 *task*.

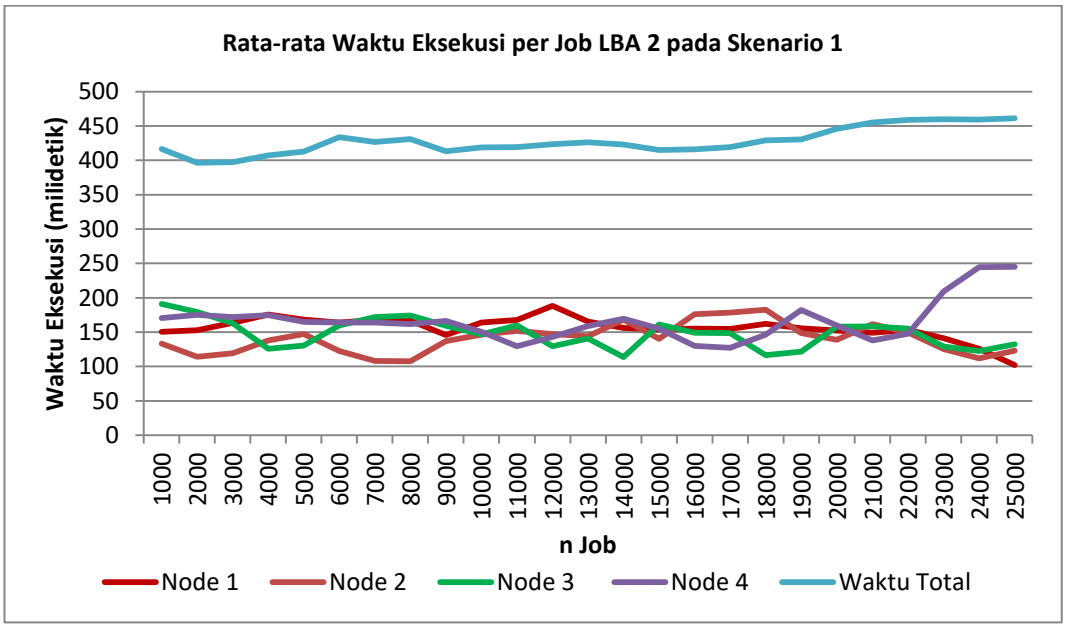
Gambar 4.5 menunjukkan hasil pengujian ditinjau dari rata-rata waktu eksekusi per *job* dan rata-rata waktu eksekusi per *bundle*. Pada hasil LBD pada Skenario 1, rata-rata waktu eksekusi *bundle* berkisar antara 100 milidetik hingga 250 milidetik. Semua *node* mengalami fluktuasi waktu eksekusi. Rata-rata waktu eksekusi *job* pada LBD adalah sebesar antara 350 milidetik hingga 420 milidetik. Perbedaan antara rata-rata waktu eksekusi *job* dengan rata-rata waktu eksekusi *bundle* berkisar antara 150 milidetik hingga 200 milidetik.



(a)

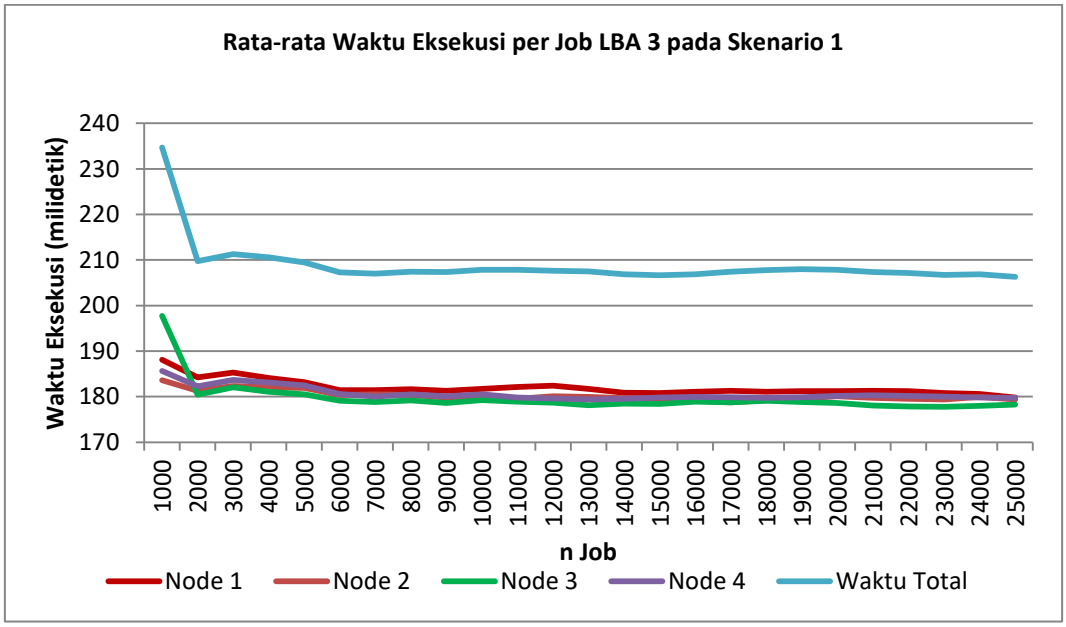


(b)



(c)

Pada hasil LBA 1, rata-rata waktu eksekusi *bundle* adalah berkisar antara 100 milidetik hingga 230 milidetik. Sedangkan rata-rata waktu eksekusi *job* berkisar antara 370 milidetik hingga 470 milidetik. Perbedaan antara rata-rata waktu eksekusi *job* dengan rata-rata waktu eksekusi *bundle* adalah berkisar antara 170 milidetik hingga 190 milidetik.



(d)

Gambar 4.5 Grafik rata-rata waktu eksekusi per *job* pada Skenario 1

Pada hasil LBA 2 pada Skenario 1, rata-rata waktu eksekusi *bundle* berkisar antara 100 milidetik hinggan 180 milidetik. Sedangkan, rata-rata waktu eksekusi *job* adalah berkisar antara 400 milidetik hingga 460 milidetik. Perbedaan antara rata-rata waktu eksekusi *job* dengan rata-rata waktu eksekusi *bundle* berkisar antara 190 milidetik hinggan 210 milidetik.

Pada hasil LBA 3 pada Skenario 1, rata-rata waktu eksekusi *bundle* berkisar antara 178 milidetik hingga 184 milidetik. Sedangkan rata-rata waktu eksekusi *job* berkisar antara 206 milidetik hingga 210 milidetik. Perbedaan antara rata-rata waktu eksekusi *job* dengan rata-rata waktu eksekusi *bundle* berkisar antara 15 milidetik hingga 20 milidetik.

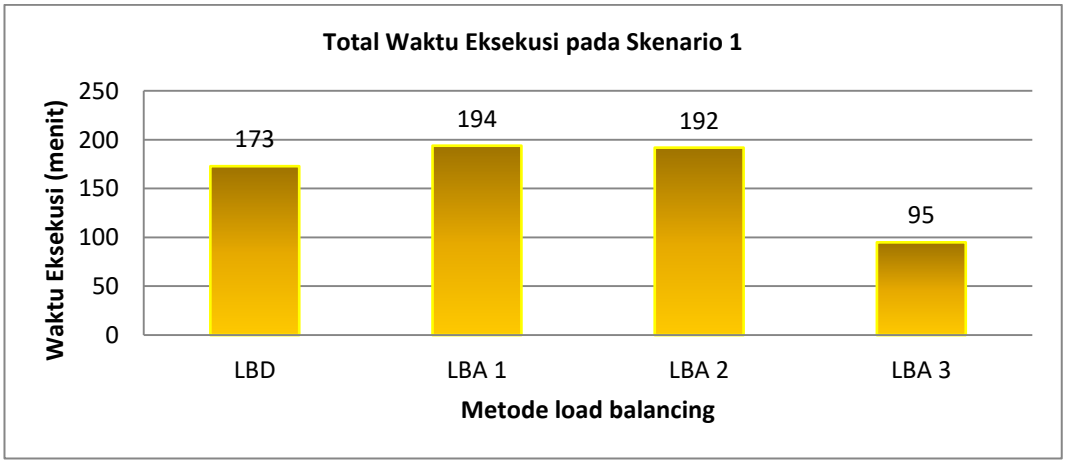
Tingginya perbedaan antara rata-rata waktu eksekusi *job* dengan rata-rata waktu eksekusi *bundle* pada LBD, LBA 1 dan LBA 2 disebabkan oleh *load balancer* yang memberikan porsi berlebih kepada beberapa *node* hingga menyebabkan *node* lain dalam keadaan *idle*. Semakin tinggi frekuensi *node idle* pada saat sistem mendapat *job*, maka akan menyebabkan total waktu komputasi juga semakin besar. Contoh kasus *node idle* pada saat sistem melakukan komputasi *job* ditunjukkan pada Tabel 4.4. Sedangkan daftar frekuensi *node idle* pada masing-masing kasus uji pada Skenario 1 ditunjukkan pada Tabel 4.5.

Tabel 4.4 Tabel sampel *node idle* pada LBD pada Skenario 1

Job	**Jumlah *task***			
	Node 1	Node 2	Node 3	Node 4
job #5	1	43	1	55
job #6	0	35	0	65
job #7	1	31	1	67
job #8	1	41	0	58
...	...	...	...	...
job #3094	0	1	81	18
job #3095	0	0	100	0
job #3096	0	1	99	0
job #3097	0	0	100	0

Tabel 4.5 Tabel frekuensi *node idle* pada Skenario 1

Metode *load balancing*	**Frekuensi *node idle***
LBD	5.034
LBA 1	24.700
LBA 2	36.203
LBA 3	59



Gambar 4.6 Grafik total waktu eksekusi pada kondisi normal

Gambar 4.6 menunjukkan perbandingan total waktu yang digunakan untuk mengeksekusi 25.000 *job* pada masing-masing kasus uji. Dari Gambar 4.6, salah satu usulan metode *load balancing* memberikan total waktu komputasi yang

paling kecil. Hal ini menunjukkan metode yang diusulkan pada LBA 3 lebih baik dibandingkan dengan metode yang digunakan pada LBD, LBA 1 maupun LBA 2.

Speedup yang diperoleh oleh LBA 3 dibandingkan dengan LBD adalah sebesar 45%, sedangkan *speedup* yang diperoleh dibandingkan LBA 1 dan LBA 2 adalah sama-sama 51%.

4.4.2. Hasil Pengujian Skenario 2

Pada pengujian Skenario 2, *node 1* diatur untuk mengalami perubahan *throughput* dan mengalami beban kesibukan selama proses pengujian. Untuk mengubah *throughput* digunakan perintah `tc qdisc` pada suatu *interface*. *Throughput* diatur berubah-ubah antara 2Mbps dan 10Mbps selama proses eksekusi berjalan. Masing-masing nilai *throughput* dijalankan selama satu menit secara bergantian.

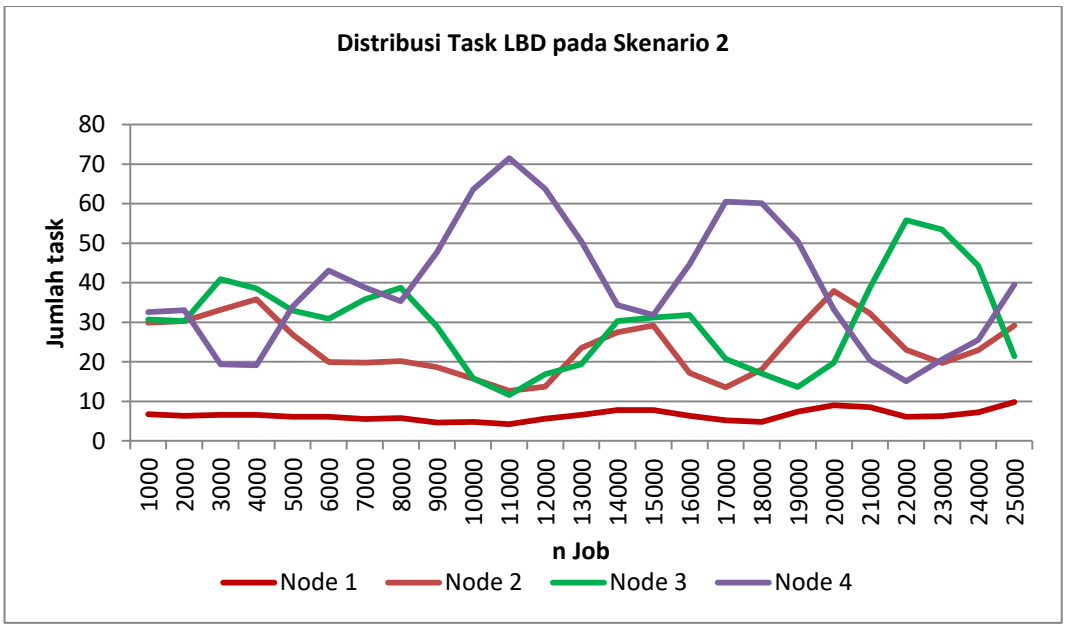
Untuk memberikan beban pada CPU, dijalankan *task* dekripsi *cipheredtext* MD5 menjadi *plaintext*. Setiap *task* menggunakan waktu komputasi selama kurang lebih 5 detik. Beban CPU sebesar 0% dan 50% diberikan secara bergantian, masing-masing satu menit. Agar eksekusi *task* berjalan selama satu menit, maka *task* tersebut dieksekusi sebanyak 12 kali. Implementasi pemberian beban kepada CPU ini ditunjukkan pada Gambar 3.14.

Uji coba dilakukan dengan memberikan 25.000 *job* kepada sistem dengan pengujian LBD, LBA 1, LBA 2 dan LBA 3. Tiap *job* berisi 100 *task* dan 100 *cipheredtext* MD5. *Task* tersebut didistribusikan ke masing-masing *node* untuk mencari *plaintext*.

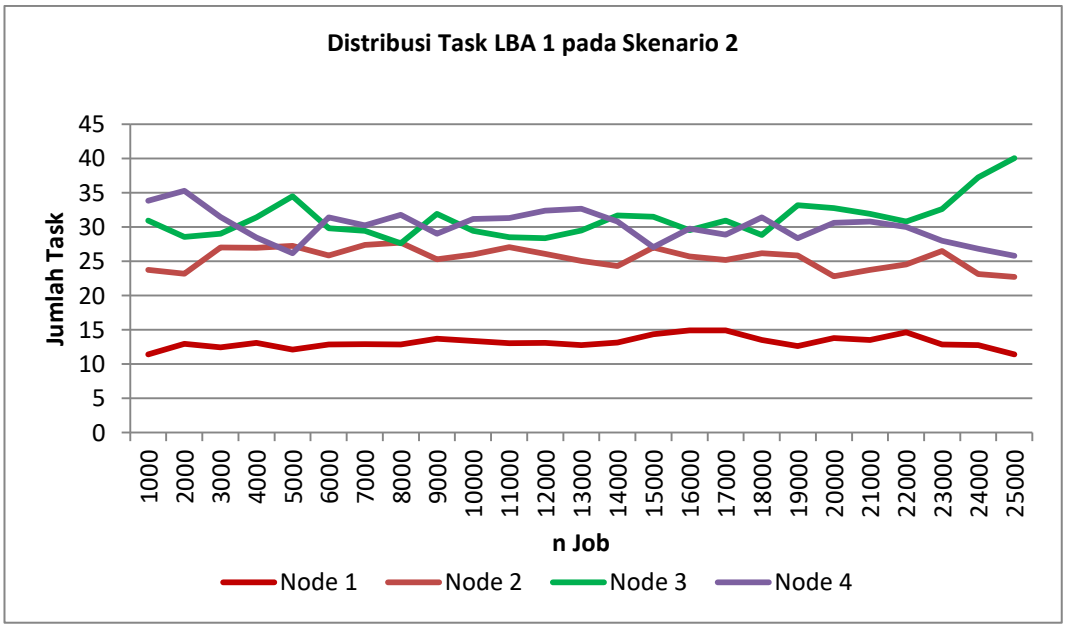
Gambar 4.7 menunjukkan hasil pengujian pada Skenario 2 ditinjau dari distribusi jumlah *task* ke masing-masing *node*. Hasil pengujian LBD, LBA 1, LBA 2 dan LBA 3 sama-sama menunjukkan fluktuasi jumlah *task* yang cukup besar. Namun, dari keempat kasus uji, LBD dan LBA 1 menunjukkan fluktuasi kecil terjadi kepada jumlah *task* yang diterima oleh *node* 1, sedangkan pada kasus uji LBA 2 dan LBA 3, *node* 1 mengalami fluktuasi jumlah *task* yang cukup besar.

Pada hasil LBD Skenario 2, sejak awal hingga akhir iterasi, *node* 1 memperoleh *task* dalam jumlah yang relatif sama, yaitu antara 5 *task* sampai 10 *task*. *Node* 1 mengalami fluktuasi yang sangat kecil. Tetapi, fluktuasi yang cukup

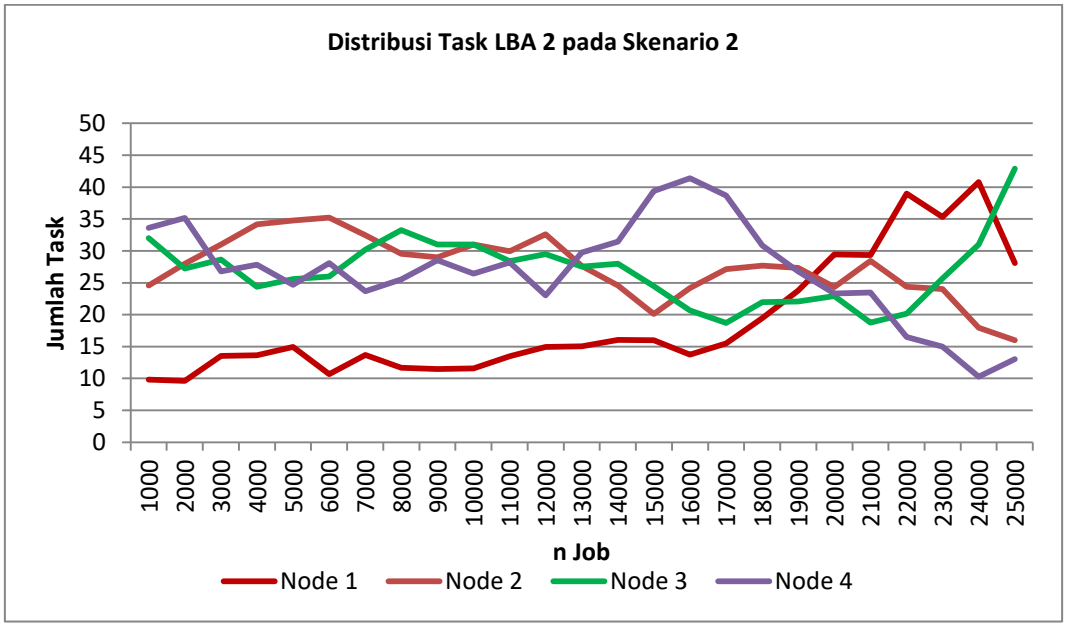
besar justru terjadi pada *node-node* yang tidak diberikan beban *throughput* ataupun beban CPU. Hal ini dapat dilihat dari jumlah *task* yang diterima oleh *node 3*. Pada awal iterasi, *node 3* memperoleh 30 *task*, kemudian pada iterasi ke-8.000 jumlah *task* yang diperoleh adalah sekitar 39 *task*, kemudian mengalami penurunan pada iterasi ke-11.000 menjadi 12 *task*, dan meningkat pada iterasi ke-16.000 menjadi 32 *task* dan pada iterasi ke-22.000 menjadi 55 *task*.



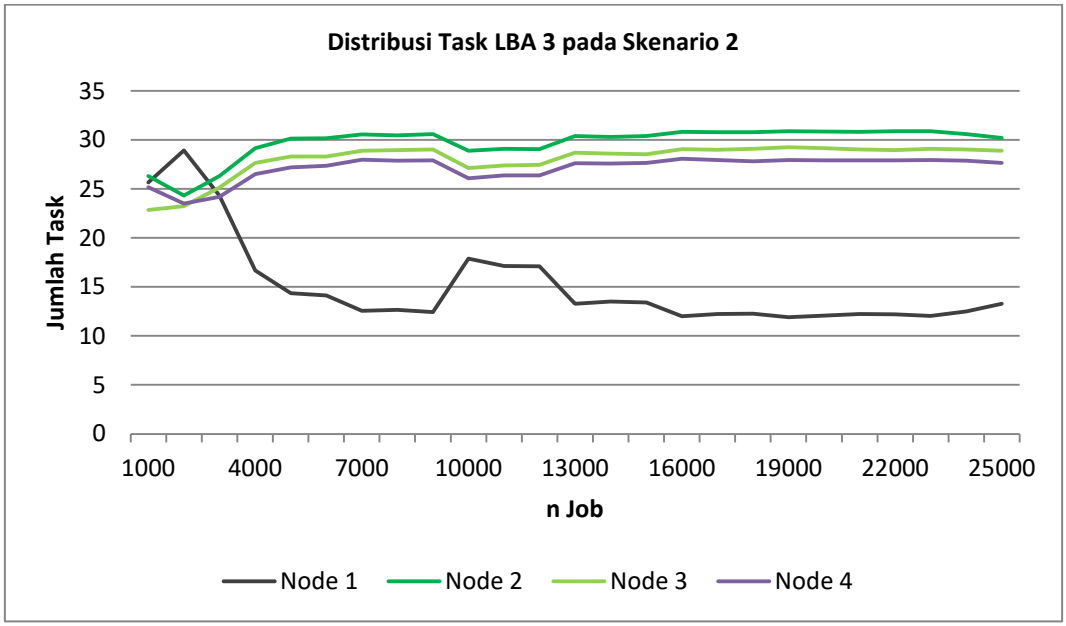
(a)



(b)



(c)



(d)

Gambar 4.7 Grafik distribusi *task* pada Skenario 2

Hampir sama dengan hasil LBD, hasil LBA 1 pada Skenario 2, pada iterasi awal mendapat sejumlah 13 *task* dan berubah-ubah antara 10 *task* dan 15 *task* hingga semua *job* selesai dieksekusi. *Node-node* yang lain tidak mengalami fluktuasi seperti yang terjadi pada LBD, namun hanya berkisar antara 24 *task*

hingga 35 *task*. Pada iterasi-iterasi akhir, *node 3* mengalami peningkatan jumlah *task* yang diterima hingga mencapai 40 *task*.

Pada hasil LBA 2 pada Skenario 2, semua *node* mengalami fluktuasi jumlah *task* yang diterima dari *server*. *Node 1*, *node* yang mengalami *throughput* berubah-ubah dan beban CPU yang berubah, pada awal iterasi mendapatkan sejumlah 10 *task*. Namun, pada iterasi-iterasi akhir, jumlah *task* yang diterima oleh *node 1* adalah sejumlah 40 *task*.

Pada hasil LBA 3 pada Skenario 2, *node 1* mengalami fluktuasi jumlah *task* yang cukup besar. Pada awal iterasi, jumlah *task* yang diterima oleh *node 1* adalah sejumlah 25 *task*. Namun, secara bertahap, jumlah *task* yang diterima mengalami penurunan hingga sejumlah 13 *task* pada iterasi-iterasi akhir. Sedangkan, *node-node* yang lain hanya mengalami fluktuasi jumlah *task* yang kecil.

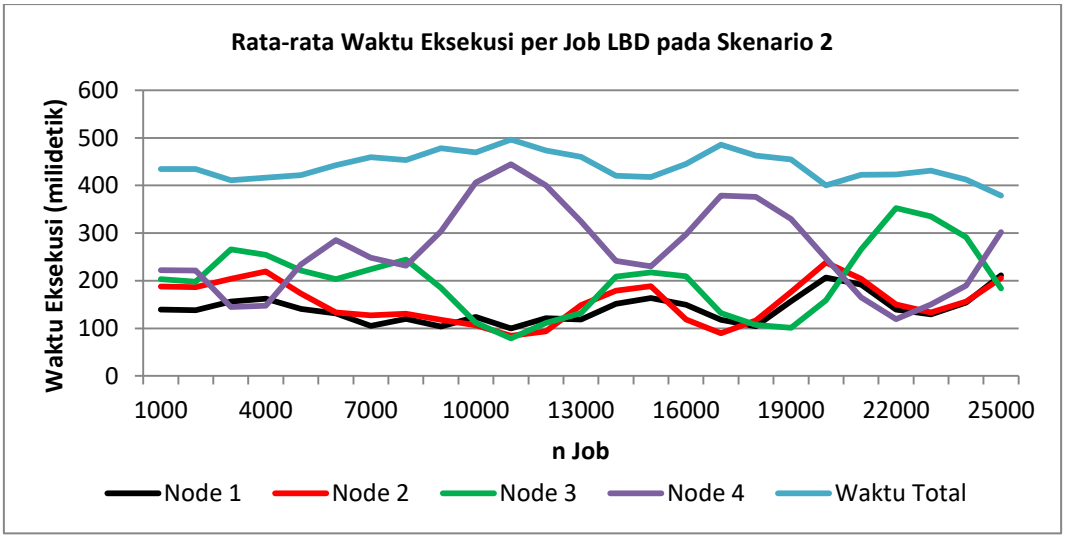
Gambar 4.8 menunjukkan hasil eksekusi Skenario 2 ditinjau dari rata-rata waktu eksekusi per *job* dan rata-rata waktu eksekusi per *bundle*. Pada hasil LBD pada Skenario 2, rata-rata waktu eksekusi *bundle* berkisar antara 100 milidetik hingga 440 milidetik. Sedangkan, rata-rata waktu eksekusi *job* berkisar antara 400 milidetik hingga 500 milidetik. Perbedaan antara rata-rata eksekusi *job* dengan rata-rata eksekusi *bundle* adalah berkisar antara 60 milidetik hinggal 120 milidetik. Pada LBD ini, rata-rata waktu eksekusi per *bundle* yang dilakukan oleh *node 1* adalah berkisar antara 100 milidetik hingga 200 milidetik. Hal ini membuat *node 1* memiliki rata-rata waktu eksekusi relatif kecil apabila dibandingkan dengan *node 3* dan *node 4* dengan rata-rata waktu eksekusi antara 100 milidetik hingga 400 milidetik.

Pada hasil LBA 1 pada Skenario 2, rata-rata waktu eksekusi *bundle* pada *node 1* berkisar antara 300 milidetik hingga 360 milidetik. Rata-rata waktu eksekusi *bundle* pada *node 1* ini selalu lebih besar dibandingkan dengan rata-rata waktu eksekusi *bundle* pada *node 2*, *node 3* maupun *node 4* yang bernilai sekitar 150 milidetik hingga 250 milidetik. Hal ini menjadikan *node 1* menjadi titik terlama yang menentukan lamanya waktu eksekusi suatu *job*. Sedangkan, rata-rata waktu eksekusi *job* adalah berkisar 450 milidetik hingga 520 milidetik. Perbedaan

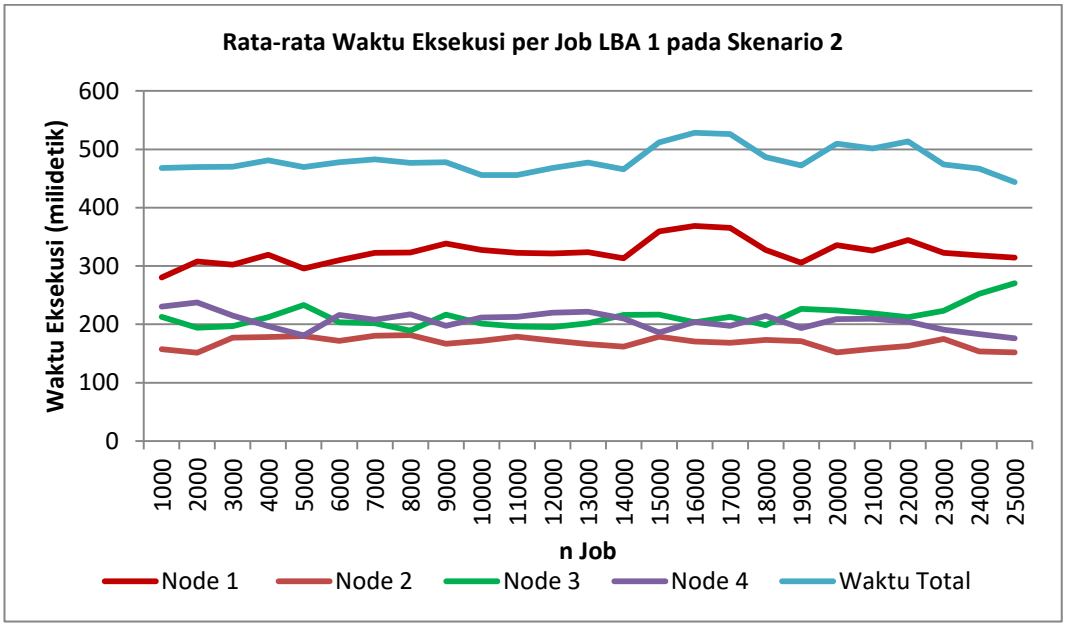
antara rata-rata waktu eksekusi *job* dengan rata-rata waktu eksekusi *bundle* adalah berkisar antara 130 milidetik hingga 200 milidetik.

Pada LBA 2 pada Skenario 2, waktu eksekusi pada *node 1* pada iterasi awal sekitar 240 milidetik. Namun, pada iterasi-iterasi selanjutnya, waktu eksekusi pada *node 1* terus mengalami peningkatan menjadi 1.000 milidetik pada iterasi ke-24.000. Hal ini berbanding lurus dengan yang ditunjukkan pada Gambar 4.7, karena pada iterasi-iterasi akhir, *node 1* mendapatkan jumlah *task* yang besar. Akibatnya, waktu komputasi *bundle* menjadi besar dan waktu eksekusi *job* juga menjadi besar. Rata-rata waktu eksekusi *job* pada LBA 2 adalah berkisar antara 550 milidetik hingga 1.160 milidetik. Perbedaan antara rata-rata waktu eksekusi *job* dengan rata-rata eksekusi *bundle* berkisar antara 160 milidetik hingga 210 milidetik.

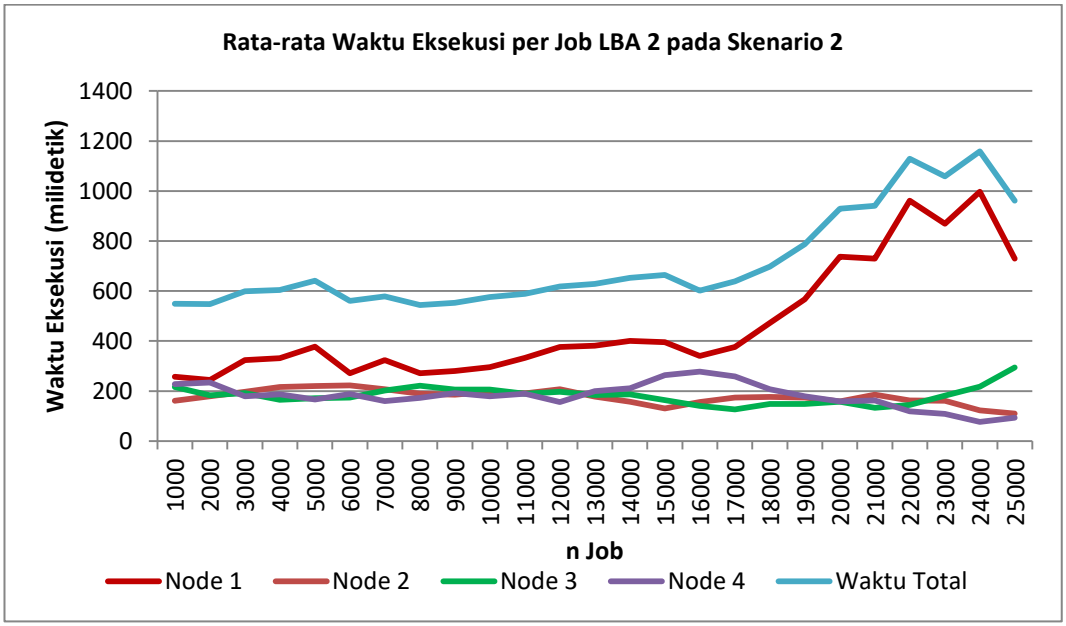
Pada hasil LBA 3 pada Skenario 2, pada awal iterasi, rata-rata waktu eksekusi *bundle* adalah sebesar 600 milidetik. Namun, pada iterasi-iterasi selanjutnya, rata-rata eksekusi *bundle* semakin menurun hingga pada kisaran 270 milidetik. Hal ini karena pada iterasi-iterasi selanjutnya, jumlah *task* yang diterima juga semakin menurun seperti yang ditunjukkan pada Gambar 4.7. Rata-rata waktu eksekusi *job* pada awal iterasi adalah sekitar 650 milidetik. Namun, seiring dengan menurunnya jumlah *task* yang diterima oleh *node 1*, maka waktu eksekusi *job* juga menjadi lebih kecil.



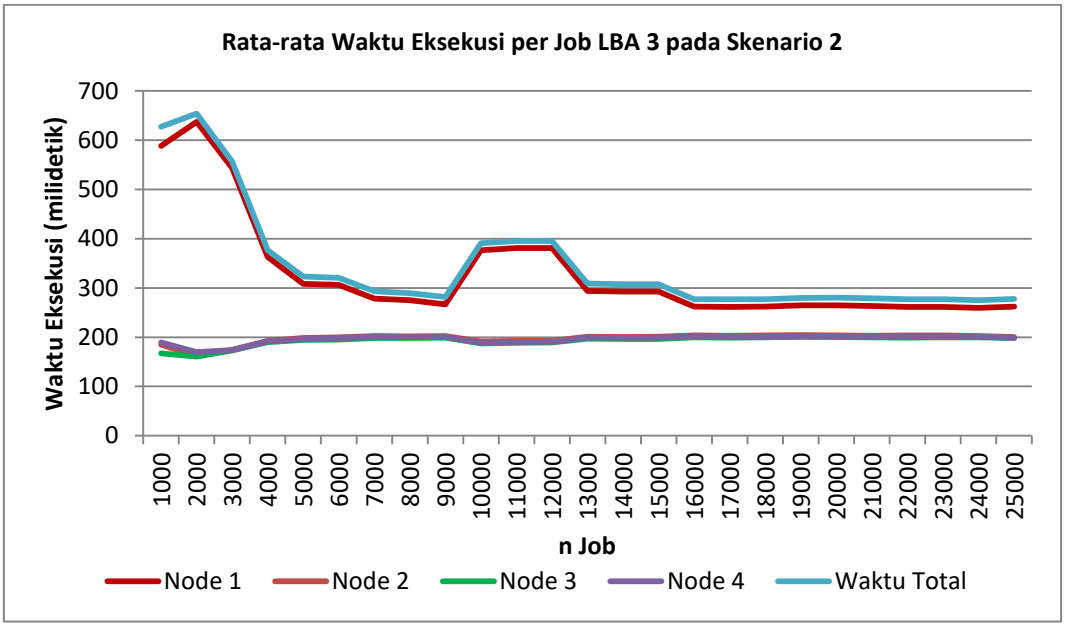
(a)


Rata-rata Waktu Eksekusi per Job LBA 1 pada Skenario 2
Waktu Eksekusi (milidetik)
600
500
400
300
200
100
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
n Job
Node 1
Node 2
Node 3
Node 4
Waktu Total

(b)


Rata-rata Waktu Eksekusi per Job LBA 2 pada Skenario 2
Waktu Eksekusi (milidetik)
1400
1200
1000
800
600
400
200
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
n Job
Node 1
Node 2
Node 3
Node 4
Waktu Total

(c)



(d)

Gambar 4.8 Grafik rata-rata waktu eksekusi per *job* pada Skenario 2

Pada LBD, LBA 1 dan LBA 2, perbandingan antara rata-rata waktu eksekusi *job* dengan rata-rata waktu eksekusi *bundle* relatif lebih besar jika dibandingkan pada LBA 3. Penyebab hal ini sama dengan yang menjadi penyebab pada Skenario 1, yaitu adanya *node idle*. Hal ini disebabkan oleh distribusi *task* yang kurang merata oleh *load balancer*. Contoh kasus *node idle* pada Skenario 2 ditunjukkan pada Tabel 4.6. Sedangkan daftar frekuensi *node idle* pada masing-masing kasus uji pada Skenario 2 ditunjukkan pada Tabel 4.7.

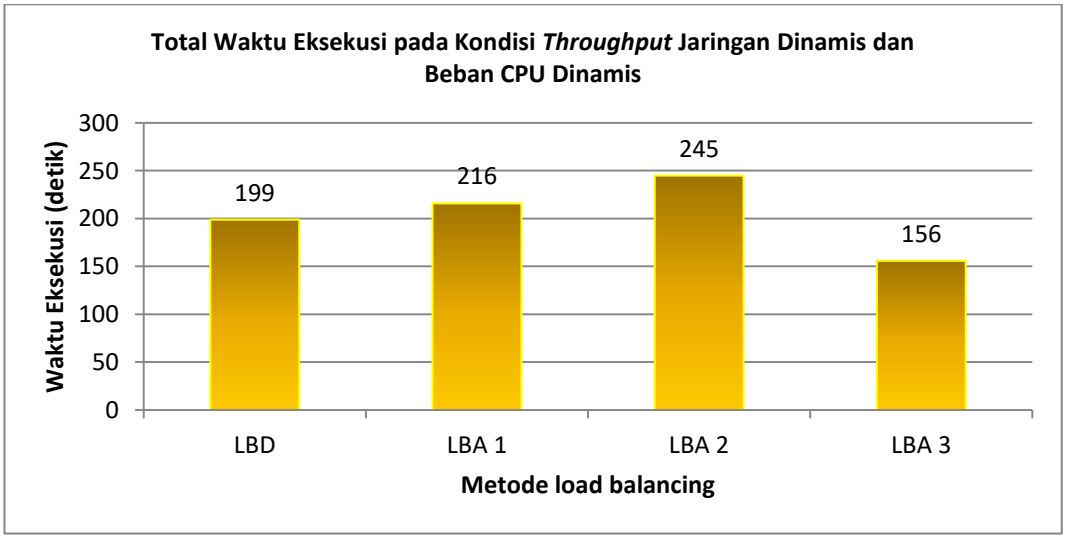
Tabel 4.6 Tabel sampel *node idle* pada LBA 1 pada Skenario 2

Job	**Jumlah Task**			
	Node 1	Node 2	Node 3	Node 4
…	…	…	…	…
job #60	42	0	55	3
job #61	0	0	56	44
job #62	43	0	1	56
job #63	41	0	5	54
job #64	0	0	52	48
…	…	…	…	…
job #3916	0	60	40	0
job #3917	0	51	0	49
job #3918	0	62	0	38

job #3919	0	0	57	43
job #3920	0	66	34	0
job #3921	0	67	33	0
job #3922	0	0	86	14
job #3923	0	76	24	0
job #3924	0	0	49	51
job #3925	0	83	0	17
job #3926	0	65	0	35
job #3927	0	0	71	29

Tabel 4.7 Tabel frekuensi *node idle* pada Skenario 2

Metode *load balancing*	Frekuensi *node idle*
LBD	4.982
LBA 1	4.255
LBA 2	29.174
LBA 3	408



Gambar 4.9 Grafik total waktu eksekusi pada Skenario 2

Gambar 4.9 menunjukkan perbandingan total waktu yang digunakan untuk mengeksekusi 25.000 *job* pada masing-masing kasus uji. Dari Gambar 4.9, salah satu usulan metode *load balancing* memberikan total waktu komputasi yang paling kecil. Hal ini menunjukkan metode yang diusulkan pada LBA 3 lebih baik dibandingkan metode yang digunakan pada LBD, LBA 1 maupun LBA 2.

Speedup yang diperoleh oleh LBA 3 dibandingkan dengan LBD adalah sebesar 21%, sedangkan *speedup* yang diperoleh dibandingkan dengan LBA 1 dan LBA 2 adalah sebesar 27% dan 36%.

BAB V
KESIMPULAN DAN SARAN

Pada Bab ini dijelaskan kesimpulan akhir yang didapatkan dari penelitian yang telah dilakukan dan juga dipaparkan saran-saran yang bersifat membangun untuk penelitian selanjutnya di masa yang akan datang.

5.1 Kesimpulan

Pengujian dan analisis yang telah dilakukan menghasilkan beberapa kesimpulan penelitian sebagai berikut:

1. Salah satu metode *load balancing* yang diusulkan pada penelitian ini, yaitu dengan menggunakan RL dengan parameter jumlah *task* (LBA 3) terbukti mampu menyeselesaikan *job* yang diberikan dalam waktu yang lebih kecil dibandingan dengan metode *load balancing* dinamis (LBD), baik ketika sumber daya jaringan dan CPU dalam kondisi tidak mengalami beban yang ditunjukkan pada Gambar 4.6 maupun dalam kondisi mengalami beban yang ditunjukkan pada Gambar 4.9.
2. Pada kondisi sumber daya jaringan dan CPU tidak mengalami beban, *load balancing* menggunakan algoritma RL dengan parameter ukuran *bundle* (LBA 3) memberikan *speedup* sebesar 45% dibandingkan dengan metode dinamis yang terdapat di dalam JPPF yang ditunjukkan pada Gambar 4.6.
3. Pada kondisi sumber daya jaringan dan CPU mengalami beban, *load balancing* menggunakan algoritma RL dengan parameter ukuran *bundle* (LBA 3) memberikan *speedup* sebesar 21% dibandingkan dengan metode dinamis (LBD) yang terdapat di dalam JPPF yang ditunjukkan pada Gambar 4.9.
4. Dari beberapa metode *load balancing* yang diusulkan, metode RL dengan parameter waktu eksekusi (LBA 1) dan metode RL dengan parameter rata-rata waktu eksekusi, rata-rata *throughput* jaringan dan rata-rata beban CPU (LBA 2), memberikan total waktu komputasi yang lebih lama dibandingkan dengan metode dinamis (LBD), sebesar 12% dan 11% pada kondisi sumber daya jaringan dan CPU tidak mengalami beban yang ditunjukkan pada

Gambar 4.6 dan sebesar 8% dan 23% pada kondisi sumber daya jaringan dan CPU mengalami beban yang ditunjukkan pada Gambar 4.9.

5.2 Saran

Dengan semakin meningkatnya penggunaan komputasi *cloud*, yang di dalamnya terdapat komputasi klaster, maka penelitian dengan topik perbaikan mekanisme *load balancing* menjadi topik yang cukup menarik. Strategi distribusi menggunakan *framework machine learning* menjanjukan memberikan keunggulan dalam menangani tugas distribusi pada lingkungan produksi yang dinamis.

Di dalam *framework* JPPF, *load balancer* bertugas untuk menentukan jumlah *task* yang akan diberikan ke suatu *node*. Setiap *node* dipilih berdasarkan metode *round Robin*. *Framework* RL dapat dikembangkan sebagai algoritma pembelajaran untuk memilih *node* mana yang mendapat prioritas mendapatkan tugas terlebih dahulu.

DAFTAR PUSTAKA

Ayyasamy, S., & Sivanandam, S. (2010). A Cluster Based Replication Architecture for Load-Balancing in Peer-to-Peer Content Distribution. *International Journal of Computer Networks & Communications*, 158-172.

Barney, B. (2016, March 15). *Introduction to Parallel Computing*. Retrieved April 26, 2016, from https://computing.llnl.gov/tutorials/parallel_comp/#Whatis

Berman, F., Wolski, R., Figueira, S., & Schopf, G. (1996). Application-Level Scheduling on Distributed Heterogeneous Networks. *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (pp. 1-28). Pittsburgh: IEEE.

Busoniu, L., Babuska, R., & Schutter, B. (2008). A Comprehensive Survey of Multiagent. *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS*, 156-172.

Choi, J., Dukhan, M., Liu, X., & Vudue, R. (2014). Algorithmic time, energy, and power on candidate HPC compute building blocks. *IPDPS'14*, 1-14.

Cohen, L. (2014). *JPPF Documentation.* Retrieved October 9, 2014, from JPPF Home: http://sourceforge.net/projects/jppf-project/files/jppf-project/jppf%204.2.2/JPPF-4.2.2-User-Guide.zip/download

Feitelson, D., & Rudolph, L. (2006). Metrics and benchmarking for parallel job scheduling. *Springer*, 1-24.

Guo, J., & Bhuyan, L. N. (2006). Load Balancing in a Cluster-Based Web Server for Multimedia Applications. *IEEE Transactions on Parallel and Distributed Systems*, 1321-1334.

He, Q., Dovrolis, C., & Ammar, M. (2005). On the Predictability of Large Transfer TCP Throughput. *SIGCOMM '05 Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, 145-156.

Lee, C., Abe, H., Hirotsu, T., & Umemura, K. (2013). Performance Implications of Task Scheduling by Predicting Network Throughput on the Internet. *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (pp. 1089-1098). IEEE.

Lee, M. (2005, January 4). *Reinforcement Learning*. Retrieved 10 9, 2014, from University of Alberta: http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node7.html

Mahato, D. P., Maurya, A. K., Tripathi, A. K., & Singh, R. S. (2016). Dynamic and Adaptive Load Balancing in Transaction Oriented Grid Service. *Green High Performance Computing (ICGHPC), 2016 2nd International Conference* (pp. 1-5). IEEE.

Microsoft. (2013, March 23). *Evaluating the Benefits of Clustering*. Retrieved October 13, 2014, from Microsoft Technet: Evaluating the Benefits of Clustering

Mirza, M., Springborn, K., Banerjee, S., Barford, P., Blodgett, M., & Zhu, X. (2009). On The Accuracy of TCP Throughput Prediction for Opportunistic Wireless Networks. *2009 6th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks.* IEEE.

Mohammadpour, P., Sharifi, M., & Paikan, A. (2008). A Self-Training Algorithm for Load Balancing in Cluster Computing. *2008 Fourth International Conference on Networked Computing and Advanced Information Management* (pp. 104-109). IEEE.

Patil, S., & Gopal, A. (2012). Need of New Load Balancing Algorithms for Linux Clustered System. *International Conference on Computational techniques And Artificial intelligence(ICCTAI'2012)*.

Rehnel, N. (2013, Januari 18). *nevik (Nevik Rehnel) - Github.* Retrieved Mei 1, 2016, from Github: https://github.com/nevik/NArmedBandit/

Schwiegelshohn, U., & Yahyapour, R. (1998). Analysis of First-Come-First-Serve Parallel Job Scheduling. *Computer Engineering Institute, University Dortmund*, 1-10.

Singh, J. P. (2005). Dynamic Load Balancing for Cluster Computing. Muenchen.

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning.* Massachusetts.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement Learning: An Introduction.* Cambrigde: MIT Press.

Sutton, R., Precup, D., & Singh, S. (197). A Framework for Temporal Abstraction in Reinforcement Learning. 1-42.

Tera, M. R., & Kota, S. (n.d.). *Infrastructure for Load Balancing on Mosix Cluster.*

Wolski, R., Spring, N., & Hayes, J. (1999). Predicting the CPU Availability of Time-shared Unix Systems on the Computational Grid. *IEEE The Eighth International Symposium on High Performance Distributed Computing*, 105-112.

Woodside, C., & Monforton, G. (1993). Fast Allocation of Processes in Distributed and Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 164-174.

Xhafa, F., & Abraham, A. (2010). Computational models and heuristic methods for Grid scheduling problems. *Future Generation Computer Systems*, 608-621.

Xiao , L., Chen , S., & Zhang, Z. (2004). Adaptive Memory Allocations in Clusters to Handle Unexpectedly Large Data-Intensive Jobs. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 577-592.

BIODATA PENULIS

Mohammad Zarkasi, biasa dipanggil Zarkasi, dilahirkan di Jember pada tanggal 11 November 1990. Penulis adalah anak kedua dari dua bersaudara. Penulis menempuh pendidikan dasar di Madrasah Ibtidaiyah Al Ma'arif Tempuran-Cakru (1997-2003), SMP Negeri 1 Kencong (2003-2006), dan SMA Negeri 1 Kencong (2006-2009). Setelah lulus SMA, penulis melanjutkan pendidikan ke jenjang perkuliahan di Jurusan Teknik Informatika Institut Teknologi Sepuluh Nopember Surabaya (2009-2013) lalu melanjutkan pada jenjang Master pada tahun 2013 di perguruan tinggi yang sama. Selama mengikuti perkuliahan, penulis aktif di Lembaga Dakwah Jurusan Keluarga Muslim Informatika (KMI) sebagai Staff Departemen Media periode 2010-2012.

Selain aktif di organisasi, penulis juga pernah menjadi asisten dosen untuk mata kuliah Pemrograman Web. Selain itu penulis pernah magang di Badan Teknologi dan Sitem Informasi – ITS selama enam bulan. Penulis juga aktif mengikuti kegiatan mentoring yang diadakan di Masjid Manarul Ilmi ITS.

Penulis dapat dihubungi melalui *e-mail* di mohammad.zarkasi@gmail.com.